64'er TOP-THEMEN AUS

SONDERHEFT 8/1985 ÖS 100,-/Sfr. 14,- Lit. 12000/hfl. 18,-/dkr. 68,- DM 14,-

Markt&Technik

64'er



ASSEMBLER

für Anfänger und Fortgeschrittene

Vollständiger Kurs zum Mitmachen mit allen notwendigen Programmen zum Abtippen ★ SMON komplett ★ Tolle Tips und Tricks ★ Viele ausführlich erklärte Beispiele

Alle Programme auch auf Diskette/Kassette erhältlich



Kurse

Bücher

Hilfsprogramme

Tips und Tricks

Tabellen

Warum nicht Maschinensprache?

Assembler und Maschinensprache: für viele ein unerhört reizvolles Wort. Denn man verbindet mit Maschinensprache sowohl Schnelligkeit als auch Flexibilität.

Wer einmal Assembler-Luft gerochen und seine ersten Programme geschrieben hat, kann sich nur noch schwer der Faszination entziehen. Es eröffnen sich Möglichkeiten, die von Basic aus nicht erreicht werden können. Dabei tun sich viele Leser noch schwer, über die ersten Schritte hinwegzukommen. Zu fremd scheinen ihnen die Befehle zu sein, zu tückisch die Fehler, die auftreten können. Mit Basic haben sie angefangen und es gibt, so sagen sie vielleicht, eigentlich keinen Grund, umzusteigen. Aber stimmt das wirklich?

Schnelligkeit ist Trumpf

Es gibt einige Gründe, weshalb nicht auf die Maschinensprache verzichtet werden kann:

Ein wichtiges Argument ist die Schnelligkeit. Manche Programme können mehrere hundertmal schneller sein als entsprechende in Basic und erlauben erst dadurch deren sinnvolle Nutzung. Stellen Sie sich nur einmal Hi-Eddi, Grafik 2000 oder Hypra-Ass in Basic vor. Allein der Gedanke daran macht schon müde.

Doch es gibt noch mehr Argumente. Manche Probleme lassen sich in Basic überhaupt nicht lösen, zum Beispiel interruptgesteuerte Programme, die Programmierung von Druckertreiber und Floppy oder der Zugriff auf den Zeichensatz des C 64.

Neuer Kurs

Viele Programmierer, die mit Basic angefangen haben, versuchen auch in Maschinensprache Fuß zu fassen. Deshalb haben wir für Sie den Assembler-Kurs »Assembler ist keine Alchimie« organisiert, der im 64'er, Ausgabe 9/84 begann und in der Ausgabe 10/85 endete. In diesem Sonderheft wird er noch einmal vollständig überarbeitet abgedruckt. Ein zweiter neuer Kurs schließt sich dem an. Er vermittelt an Hand von vielen Programmen und Beispielen, mit einer Menge Tips und Tricks, Programmiertechniken für Anfänger und Fortgeschrittene. Falls Sie noch keinen Assembler oder einen Maschinensprache-Monitor besitzen – wir drucken hier noch einmal einen kompletten, erweiterten »Werkzeugsatz« ab: Hypra-Ass, SMON mit allem drum und dran und einen zu Hypra-Ass kompatiblen Reassembler. Mit diesem Programmpaket sind Sie in der Lage, jede noch so kniffelige Aufgabe in Maschinensprache optimal zu lösen.

Tips und Tricks für Anfänger und Fortgeschrittene

Wir geben Ihnen viele Hinweise und Hilfen für den Einstieg und zeigen Ihnen, wie man mit einem Assembler arbeitet, was Makros sind und wie sie eingesetzt werden.

Natürlich fehlen auch die Listings nicht, alle mit dokumentiertem Quellcode, damit Sie nachvollziehen können, wie bestimmte Probleme in Maschinensprache gelöst werden.

Am Schluß dieses Sonderheftes haben wir noch einen Leckerbissen für Sie. Die wichtigsten Tabellen zusammengefaßt, die Sie sonst aus mehreren Heften und Büchern zusammensuchen müßten und zum Teil nirgends in dieser Form finden. (gk)

Disketten und Kassettenservice

Wer keine Zeit oder keine Lust hat, alle Programme selbst in mühevoller Kleinarbeit abzuschreiben, kann wieder auf den bewährten Disketten- und Kassettenservice zurückgreifen. Alle Programme, die mit dem Diskettensymbol im Inhaltsverzeichnis gekennzeichnet sind, gibt's auf Diskette. Wenn sie mit dem Kassettensymbol versehen sind, auch auf Kassette.

Wichtiger Hinweis:

Einige auf Kassette angebotenen Programme sind vom Autoren auf die Benutzung des Floppy-Laufwerks abgestimmt. Für den Einsatz mit der Datasette müssen einige Programmteile von Ihnen selbst verändert werden. Wir sind gerne bereit, Informationen über entsprechende Anpassungen im 64'er zu veröffentlichen, beziehungsweise Kontakte zu vermitteln. Wir können jedoch leider nicht bei den einzelnen Anpassungen behilflich sein.

Disketten: Bestellnummer L6 85 S8D / 29,90 Mark
Kassetten: Bestellnummer L6 85 S8K / 19,90 Mark

Inhalt

Assembler ist keine Alchimie

Den kompletten Assembler-Kurs in einem Stück wünschten sich viele 64'er-Leser. In diesem Sonderheft können wir diesen Wunsch realisieren. Der Kurs soll nicht unbedingt ein Buch über Maschinensprache ersetzen, er wird Ihnen jedoch helfen, diese Sprache leichter zu verstehen.

Vermutlich hat es Ihnen auch schon ab und zu in den Fingern gejuckt, wenn Sie von Wunderdingen gelesen haben, die man per Maschinensprache mit dem Computer machen kann. Vielleicht haben Sie sogar schon mal nichtsahnend angefangen einzutippen, was Sie als Assemblerlisting sahen. Doch schon nach dem ersten »C000 LDA # $00« und RETURN weigerte sich der Computer mit einem lapidaren »SYNTAX ERROR«. Wieso, werden Sie sich gefragt haben, das ist doch nun die Sprache unserer Maschine, nämlich Maschinensprache, was habe ich falsch gemacht?

Dann sind Sie sicherlich mal auf diese merkwürdigen Basic-Programme gestoßen, in denen ein langer Wurm von DATA-Zeilen mit einem kleinen FOR..NEXT.. POKE-Kopf vorne und einem SYS-Schwanz hinten enthalten ist, und die man Basic-Lader nennt. Sie haben fleißig Zahlen eingetippt – das Ganze hoffentlich sofort gespeichert-, vorschriftsmäßig mit dem SYS-Befehl gestartet und auf einen scheintoten Computer geschaut, der nur noch durch Aus- und Einschalten wiederzubeleben war. Wenn Sie dann nach langer Fehlersuche den irrtümlich eingetippten Punkt durch ein Komma ersetzt haben (oft finden Sie auch keinen Fehler, denn bei langen DATA-Sequenzen schlägt der Druckfehlerteufel mit Vorliebe zu), werden Sie sich gefragt haben, warum in aller Welt dieses kleine Mißgeschick den ganzen Computer abstürzen läßt. Sie merken vermutlich schon, daß mir das alles und noch mehr (worüber ich schamhaft schweige) passiert ist. Die Konsequenz war, daß ich losging, um ein schlaues Buch zu erwerben. Aber merkwürdig, damals tauchte der Begriff »Maschinensprache« in keinem Titel auf. Irgendwann begriff ich, daß Assembler und Maschinensprache irgend etwas miteinander zu tun haben.

Aber da fing das ganze Elend erst richtig an: Da gab es 6502-, Z80-, 8080-, 8085-, 6800-Assembler, da waren irgendwelche Schaltpläne, anscheinend, wie man wo was hinlötet- für mich als Nichtelektroniker eine Art moderner Kunst-, da war von CPU, Bussen, negativen Flanken, Zweiphasentakten die Rede.

Ich habe mich furchtbar geärgert über die Geheimsprache, die es dem Uneingeweihten verwehrt, etwas zu verstehen. Seither hat sich einiges verändert. Die Geheimnisse sind keine mehr und ich werde Ihnen in dieser Serie ohne verschlüsselte Sprache die magischen Zirkel der Assembler-Alchimisten offenbaren. Heute gibt es auch Bücher über »Maschinensprache auf dem Commodore 64« und es sei Ihnen angeraten, ruhig auch das eine oder andere durchzuarbeiten. Sie werden allerdings feststellen, daß die meisten davon gerade dort aufhören, wo es anfängt spannend zu werden: bei der Benutzung von Routinen des Betriebssystems

und des Interpreters. Deswegen soll der Schwerpunkt dieses Artikels woanders liegen:

Wir werden das notwendige Grundwissen über die Hardware nur ganz knapp behandeln, dann das Vokabular des 65xx-Assemblers kennenlernen. Den Hauptteil des Artikels verbringen wir aber mit Dingen, über die es kaum Literatur gibt, nämlich wie man für eine Unzahl von Programmieraufgaben nicht nochmal das Rad erfinden muß, weil es schon längst in unserem Computer existiert.

Bevor wir loslegen, will ich Ihnen noch etwas Literatur empfehlen:

a) Wenn wir über Speicheraufbau, das binäre und das hexadezimale Zahlensystem reden, sollten Sie die Serie »Reise durch das Wunderland der Grafik« gelesen haben, die in der 64'er in den Folgen 1 und 2 (Ausgaben 4/84 und 5/84) diese Themen grundlegend behandelt hat. (Auch als Buch unter gleichnamigem Titel erschienen.)

b) Als Nachschlagebuch sehr wertvoll ist das Buch von Raeto West: C 64 Computer Handbuch. Hier finden Sie auch viele Tips und Tricks.

c) Später wird Ihnen dieses Buch fast unentbehrlich vorkommen: R. Babel, M. Krause, A. Dripke: Systemhandbuch zum Commodore 64 (und VC 20), München 1983

Weitere Literaturempfehlungen werde ich Ihnen von Fall zu Fall geben und Sie finden sie auch in der Bücherecke. Gerade zu unserem Computer erscheint fast jeden Monat ein neues Buch und es ist nicht einfach, die Spreu vom Weizen zu trennen.

1. Einige Begriffsklärungen

Zunächst einmal muß ich Sie enttäuschen: Ich glaube kaum, daß Sie mit Ihrem Computer je einmal in Maschinensprache verkehren werden! Maschinensprache, das ist die einzige, die der Computer direkt versteht, das sind vorhandene oder nicht vorhandene Stromimpulse oder Magnetisierungszustände, die bei unserem Computer durch 8-Bit-Binärzahlen auszudrücken sind. Was wir mit unserem Computer reden werden ist Assembler. Mit dem Computer sprechen soll heißen: Mit dem Gehirn unseres Computers, dem Prozessor, oft auch CPU (von Central Processing Unit=Zentraler Arbeitsbaustein) genannt, verkehren, also ihm Befehle zu geben. Solche CPUs werden bei verschiedenen Firmen hergestellt, sind daher unterschiedlich aufgebaut und auch unterschiedlich ansprechbar. Ein weit verbreiteter Prozessortyp ist der 6502, der das Gehirn des C 64 und auch des VC 20 ist. Genau genommen ist das Gehirn des C 64 allerdings der 6510, ein dem 6502 fast identischer Prozessor. Auf den kleinen Unterschied werden wir noch zu sprechen kommen. Beide (6502 und 6510) sind in 6502-Assembler zu programmieren und wenn wir diese Sprache sprechen, sind für uns alle 6502-Computer zugänglich: Commodore, Apple, Atari und einige andere. Nun wissen Sie aber immer noch nicht, was Assembler eigentlich ist. Das englische Wort »assemble« heißt auf deutsch etwa montieren, zusammenstellen. Es handelt sich also um eine Programmiersprache und weil sie sehr eng am Computer orientiert ist, spricht man von einer »maschinenorientierten« Programmsprache im Gegensatz zu »problemorientierten« Programmsprachen wie Basic, Pascal, Cobol etc., die – so sollte es jedenfalls sein – auf jedem Computertyp gleich aussehen.

Ein Assembler ist aber noch etwas anderes, nämlich ein Software-Instrument, das einen in Assembler geschriebenen Befehl in die Maschinensprache übersetzt. Man spricht vom Vorgang des Assemblierens. Das umgekehrte leistet ein Disassembler, welcher uns Maschinensprache durch Rückübersetzung lesen hilft. Um die Verwirrung noch etwas zu steigern, sage ich Ihnen auch noch, was ein Monitor ist. In diesem Zusammenhang ist kein Bildschirmgerät damit gemeint, sondern ebenfalls ein Software-Instrument, das den Einblick in die Register und Speicher des Computers gewährt.

Damit Sie nun den Überblick völlig verlieren, sei abschließend zu diesem Sprachenwirrwarr noch erzählt, daß Software-Pakete, die sowohl Assembler als auch Disassembler als auch Monitor enthalten und noch eine Menge anderer brauchbarer Dinge, oft als »Assembler« angeboten werden. Das ist ein alter Trick der Alchimisten, verschiedenen Dingen den gleichen Namen zu geben!

2. Basic contra Assembler

Um das Nachfolgende deutlich zu machen, schalten Sie bitte Ihren Computer an und tippen die beiden folgenden Programme ein, die beide genau dasselbe tun: Das obere Viertel unseres Bildschirmes mit dem Buchstaben A füllen (beim VC 20 ist es die obere Hälfte). Zunächst einmal in Basic:

```
10 FOR I=1024+255 TO 1024 STEP-1
20 POKE I, 1:POKE I+54272,14
30 NEXT I
```

Für den VC 20 (Grundversion und 3-KByte-Erweiterung) ist zu setzen: anstelle von 1024 jetzt 7680, statt 54272 jetzt 30208 und statt 14 die 6. Wenn Sie mehr als die 6,5 KByte im VC 20 haben, dann setzen Sie statt 1024 jetzt 4096, statt 54272 jetzt 34304 und ebenfalls statt 14 die 6. Das Programm braucht 55 Byte + 7 Byte für die Variable I, macht zusammen 62 Byte Speicherplatz. Es geht ganz schnell und wenn Sie es schaffen, können Sie ja mal mitstoppen, wie lange es von RUN bis READY braucht: zirka 4 Sekunden.

Jetzt dasselbe in Assembler. Weil wir aber noch nicht soweit sind, erst mal als Basic-Lader, der uns das Programm in den Speicher bringt (wir kommen dazu gleich noch). Geben Sie also NEW ein und dann:

```
10 FOR I=7000 TO 7000+16
20 READ A:POKE I,A:NEXT I :END
30 DATA 160,255,162,14,169,1,153,255,
3,138,153,255,215,136,208,244,96
```

Beim VC 20 geben Sie bitte statt der 14 (Zeile 30,4.Zahl) eine 6 ein. Starten Sie den Basic-Lader mit RUN und nach dem READY geben Sie NEW und CLR ein: wir brauchen ihn nicht mehr. Ab Speicherstelle 7000 steht jetzt unser Assemblerprogramm als Maschinencode. Daß es wirklich dasselbe tut wie das Basic-Programm erfahren Sie durch SYS 7000. Da hatten Sie vermutlich gar keine Zeit mehr, auf die Stoppuhr zu drücken! (5,4 Millisekunden etwa dauert das ohne die Zeit, die der Basic-Interpreter für den Befehl SYS benötigt). Außerdem braucht das Programm 17 Byte Speicherplatz.

Genau das ist es, was die Assemblerprogrammierung so reizvoll macht: Der Speicher faßt mehr an Programm und die Ausführung des Programmes geht fast 1000mal so schnell! Dazu kommen natürlich noch einige andere Kriterien, denn viele Probleme sind zum Beispiel in Basic nicht lösbar, sondern nur mit dem vielseitigeren Assembler.

Unser Computer ist darauf vorbereitet, daß wir ihn in Basic ansprechen. Er enthält im Normalfall sofort nach dem Einschalten ein stets präsentes Übersetzungsprogramm, den Interpreter, welcher unsere Basicanweisungen für ihn verständlich interpretiert. Auch das ist ein Unterschied zu Assemblerprogrammen: Ist ein solches Programm erst einmal assembliert (also als Maschinensprache im Speicher vorhanden), braucht man kein Übersetzungsprogramm mehr. Basic-Programme dagegen müssen bei jedem Durchlauf von vorne bis hinten ständig übersetzt werden, sie laufen nicht ohne vorhandenen Interpreter. Wie so ein Interpreter im Prinzip arbeitet und was ihn von einem sogenannten Compiler

unterscheidet, können Sie im 64'er, Ausgabe 4/84 und im 64'er Sonderheft 6 (Top-Themen) im Artikel von M. Törk über seinen Strubs-Precompiler nachlesen.

Dort sehen Sie dann auch, daß ein Compiler zwar ein Basic-Programm enorm beschleunigen kann, aber bei weitem nicht an die Geschwindigkeit reiner Assemblerprogramme heranreicht, vom Speicherplatzbedarf ganz zu schweigen.

3. Wie sag ich's meinem Computer?

Leider haben weder der C 64 noch der VC 20 einen Assembler implementiert. (Sie merken, daß jetzt von dem Software-Paket die Rede ist!). Es gibt einen etwas mühseligen Weg, dieses Handicap zu umgehen: den Basic-Lader. Wie ist also der Weg, mit einem solchen Lader eigene Maschinenprogramme in den Computer zu bekommen?

a) Erstellen des Assemblerprogrammes. Das zu lernen ist die Hauptaufgabe in diesem Artikel. Das Ergebnis wird eine Kette von Befehlen sein, zu denen zum Beispiel der Befehl RTS gehört.

b) Jedem Befehl in Assembler entspricht in Maschinensprache ein Binärcode in einer Speicherstelle. Diese Codes sind in Listen nachschlagbar: RTS entspricht dem Binärcode 0110 0000.

c) Der Code muß in eine Speicherstelle eingegeben werden. Das geschieht von Basic aus mit dem POKE-Befehl. Weil aber Basic keine Binärzahlen kennt, muß der Code ins Dezimalsystem umgerechnet werden. Glücklicherweise sind in den Tabellen meist schon die Codes als Dezimal- oder wenigstens als Hexadezimalzahlen enthalten. RTS ist dezimal 96 (oder hexadezimal 60, das auch $ 60 geschrieben werden kann). Man POKEt nun an die richtige Adresse den Wert 96, also zum Beispiel POKE 7016,96

d) Auf diese Weise wird Byte für Byte in der Programmabfolge verfahren. Das reine POKEn geschieht dann eben in der Form wie im oben gezeigten Basic-Lader. Mühsam, mühsam! Auch kann man leider nur mit dem PEEK-Kommando nachsehen, was denn nun im Speicher steht (PEEK (7016) gibt uns den Wert 96, entsprechend RTS).

Ein anderer Weg ist, den in diesem Sonderheft abgedruckten »SMON« abzutippen, oder sich die Leser-Service-Diskette zu bestellen.

Assembler (das Software-Paket) gibt es in den unterschiedlichsten Ausführungen. Es gibt beispielsweise Direkt-Assembler, die jede Programmzeile sofort nach dem RETURN assemblieren, aber auch 2-Pass-Assembler, bei denen das erst nach Abschluß des Programms insgesamt durch einen Befehl (zum Beispiel ASSEMBLE) geschieht. Bei einigen kann man (ähnlich wie bei Basic mit REM) Kommentare anfügen, bestimmten Programmstellen Namen geben (LABEL), ganze Programmabschnitte mit einem Merknamen aufrufen (MAKROS) und so weiter. Was Sie für sich bevorzugen, bleibt Ihnen natürlich überlassen. Die in diesem Artikel beschriebenen Programme werden am Anfang auf diese schönen Erleichterungen verzichten, es wird sozusagen der nackte Assembler verwendet. Was Sie aber außer dem reinen Assembler noch brauchen, ist ein Disassembler und ein Monitor (ich habe schon erklärt, welchen ich meine), damit wir unseren Computer (fast) immer im Griff haben.

4. Wie funktioniert unser Computer?

Weil das Programmieren in Assembler einen viel engeren Kontakt zu technischen Einzelheiten unseres Computers erfordert, ist es notwendig, ein wenig über diese Innereien und ihre Funktion zu wissen. Sehen Sie sich dazu bitte das Bild 1 an.

Da sehen wir zunächst unseren Mikroprozessor, der meist eine Menge Funktionen in sich vereinigt (dazu kommen wir noch). Im Prinzip ist das unsere CPU (Zentraler Arbeitsbaustein). Der Prozessor steht über eine Reihe von Leitungen mit dem Rest des Computers in Verbindung. Diese Leitungen werden im Fachjargon BUSSE genannt. Da ist zunächst einmal der sogenannte Adreßbus, auf dem 16-Bit-Adressen transportiert werden, die der Prozessor erzeugt, und die die Herkunft oder auch das Ziel von Daten festlegen, die über den Datenbus laufen. Dieser kann 8-Bit-Daten transportieren, und zwar schreibend oder lesend, also zum Beispiel vom Prozessor zum RAM (schreibend), vom RAM zum Prozessor (lesend) und so weiter. Außerdem gibt es da noch einen Steuerbus, der verschiedene Synchronisationsaufgaben durchführen hilft. Links vom Prozessor ist ein Taktgeber angedeutet. Damit nichts durcheinander kommt, läuft alles im Computer sozusagen im Gleichschritt. Diese Uhr ist gewissermaßen der Trommler, den Sie vielleicht von den alten Ruder-Galeeren kennen. Dann sehen Sie einen ROM-Bereich, also einen Nur-Lese-Speicher (Read Only Memory). Daß man hier nur herauslesen kann, ist durch den Pfeil zum Datenbus

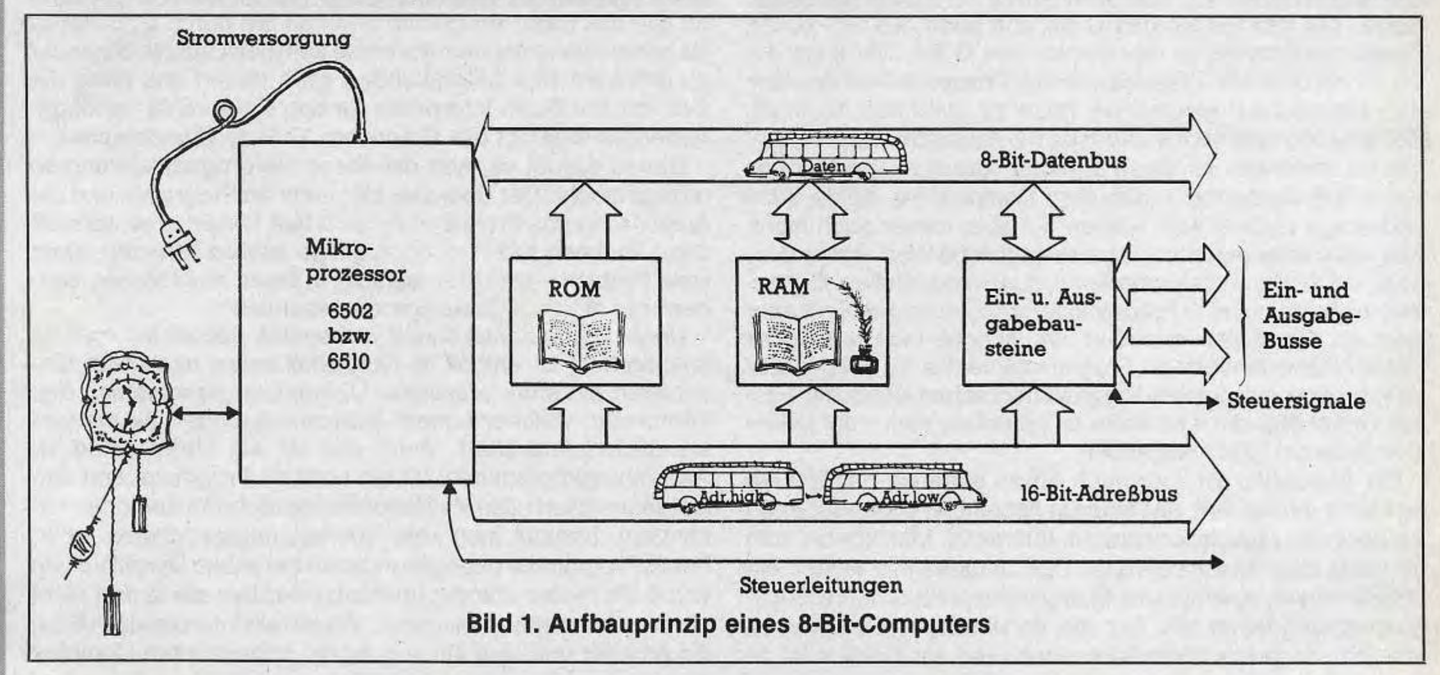


Bild 1. Aufbauprinzip eines 8-Bit-Computers

gekennzeichnet. Doppelpfeile finden wir aber beim RAM (Random Access Memory), einem Speicher für beliebigen Zugriff, also lesend und schreibend, und bei den Ein- und Ausgabebausteinen, die den Kontakt des Computers mit der übrigen Welt erlauben, also auch mit uns. Dieses Aufbauprinzip finden wir bei allen 8-Bit-Computern.

5. Das Innenleben eines Mikroprozessors

Um es gleich nochmal zu sagen: Was hier erzählt wird, ist nicht dazu geeignet, Elektronik-Freaks den totalen Durchblick zu geben. Wenn Sie das aber gerne möchten, dann sehen Sie sich zum Beispiel die Blockschaltbilder an im »Programmer's Reference Guide« für den Commodore 64 auf Seite 404 oder im »MOS-Hardware-Handbuch« auf Seite 34. Auch Rodney Zaks' Buch »Programmierung des 6502« ist zu empfehlen. Er hat sich viel Mühe gegeben, sich verständlich auszudrücken. Mir kommt es nur auf den allgemeinen Überblick an. Den sollen Sie bekommen, wenn wir uns jetzt zusammen Bild 2 betrachten.

Da sehen Sie zunächst als Herzstück des Prozessors, die ALU (Arithmetik Logical Unit), also den arithmetisch-logischen Baustein. Die ALU hat die Fähigkeit, Rechenoperationen auszuführen mit Daten, die sie über den Datenbus und normalerweise vom Akkumulator erhält. Das Ergebnis wird ebenfalls im Akkumulator abgelegt (daher auch der Name: von akkumulieren, etwas ansammeln). Der Akkumulator ist das Register, das uns als Programmierer am häufigsten beschäftigen wird. Er ist die Sammel- aber auch die Verteilerstelle für fast alle Daten, die wir hin- und herschieben wollen. Sowohl der Akku (so werde ich ihn, in der Hoffnung auf Ihr wohlwollendes Verständnis, künftig bezeichnen) als auch alle anderen Register, das heißt, die höchste Zahl, die darin bearbeitet werden kann, ist 255 (binär 1111 1111). Nahezu ebenso oft wie den Akku werden wir die beiden sogenannten Index-Register X und Y benutzen. Warum man sie Index-Register nennt, werden Sie noch im Verlauf des Kurses sehen. Als nächstes zum Prozessor-Statusflaggen-Register (hier P genannt). Man findet darin angezeigt, ob eine Rechenoperation ein negatives Ergebnis hatte, ob eine Null aufgetaucht ist oder ob ein Übertrag stattgefunden hat. Auch dieses Register wird uns noch häufig begegnen. Das Stapelregister, auch Stackpointer (Stapelzeiger) genannt, gibt uns Auskunft über den Füllungsgrad eines 256 Byte großen speziellen Speichers, der vom Prozessor direkt verwaltet wird. Auch damit werden wir noch oft zu tun haben. Schließlich kommen wir zur vorhin erwähnten Ausnahme, zum Programmzähler (PCL, PCH). Das ist ein 16-Bit-Register, das sich aus zwei 8-Bit-Registern (PCL für das LSB und PCH für das MSB) zusammensetzt und daher alle 65535 Speicherplätze ansprechen kann. Hier ist immer die Adresse des nächsten abzuarbeitenden Befehls enthalten.

Ich will an dieser Stelle nicht in die Einzelheiten der Befehlsabarbeitung einsteigen (das können Sie auch bei Rodney Zaks nachlesen, wenn Sie es genau wissen wollen). Es soll nur gesagt sein, daß sich die Verarbeitung in drei Schritte unterteilen läßt:

a) den nächsten Befehl holen
b) den Befehl decodieren
c) den Befehl ausführen

Zu c) ist noch zu sagen, daß es Befehle gibt, die der Prozessor ohne weitere Angaben ausführen kann. Für andere müssen erst noch weitere Daten aus dem Speicher geholt oder dort abgelegt werden. Deswegen brauchen die Befehle unterschiedliche Zeiten zur Ausführung. Die Zeit wird als Anzahl von sogenannten Taktzyklen in den Befehlstabellen angegeben. Unser Computer hat eine Taktfrequenz von rund 1 MHz, was bedeutet, daß ein Taktzyklus etwa eine Mikrosekunde (10^{-6} Sekunden) dauert. Auf diese Weise wurde die Zeitdauer für unser kleines Demonstrationsprogramm zu Anfang berechnet. Auch das werden Sie noch lernen.

6. Der Speicher unseres Computers: eine Straße mit 65536 Hausnummern

Dieser Artikel ist für den VC 20 und den C 64 geschrieben. Den Speicheraufbau des Commodore 64 finden Sie in der April-Ausgabe '84 dieser Zeitschrift ab Seite 119. Deswegen soll hier nur der des VC 20 gezeigt werden. Man muß beim VC 20 zwei Konfigurationen unterscheiden — sehr zum Leidwesen der Benutzer. In Bild 3 ist die Aufteilung gezeigt, die in der Grund- und der um 3 KByte erweiterten Version vorliegt.

In Bild 4 sehen Sie die Speicheraufteilung, die bei mehr als 6,5 KByte eingestecktem Speicher gültig ist.

Wenn Sie die VC 20 Speicherarchitekturen mit der des C64 vergleichen, werden Sie eine Reihe von Unterschieden

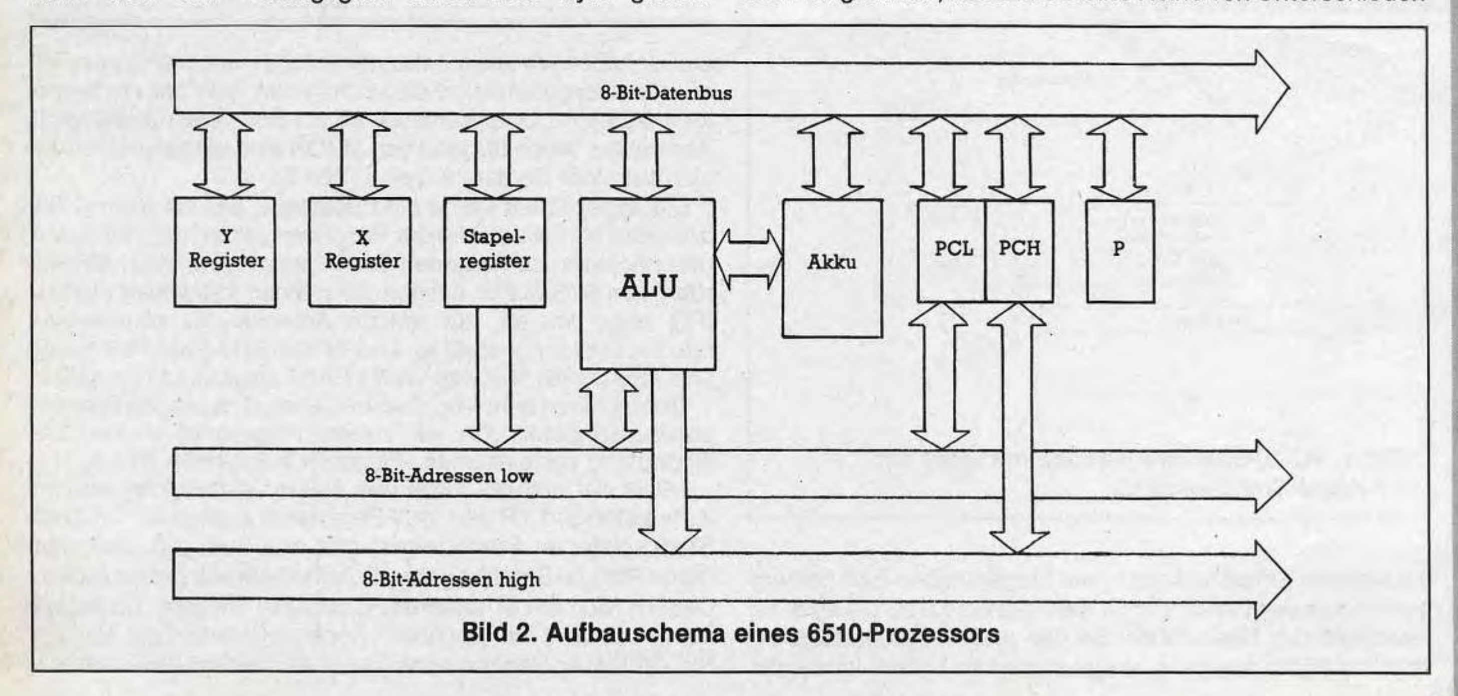


Bild 2. Aufbauschema eines 6510-Prozessors

feststellen. Genau besehen gibt es an den wichtigen Punkten aber eine Menge Gemeinsamkeiten! Der VC 20 kennt nur Speicher-Häuser mit Erdgeschoß, im Gegensatz zum C 64,

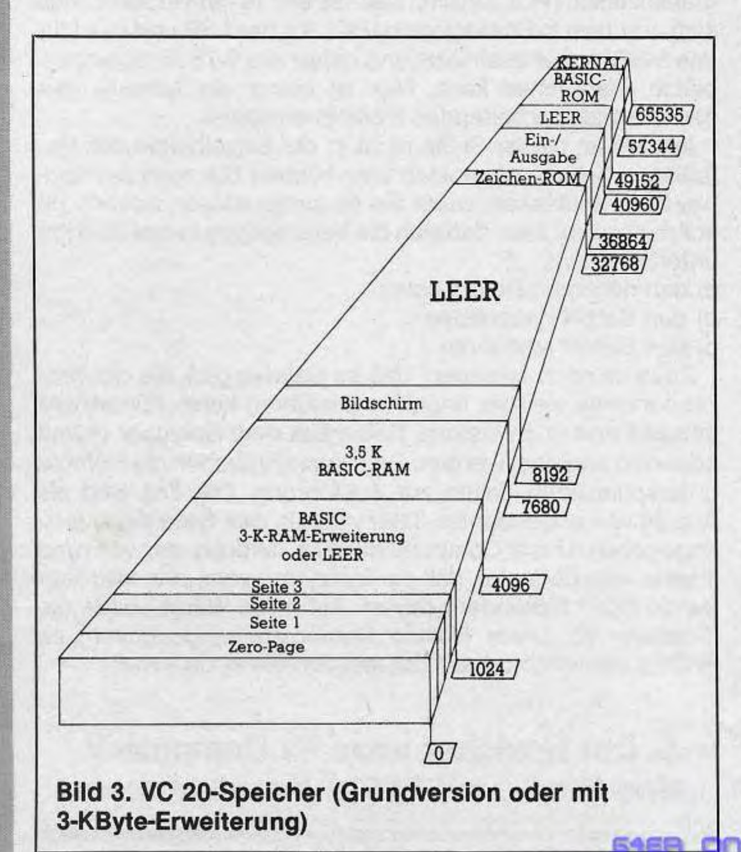


Bild 3. VC 20-Speicher (Grundversion oder mit 3-KByte-Erweiterung)

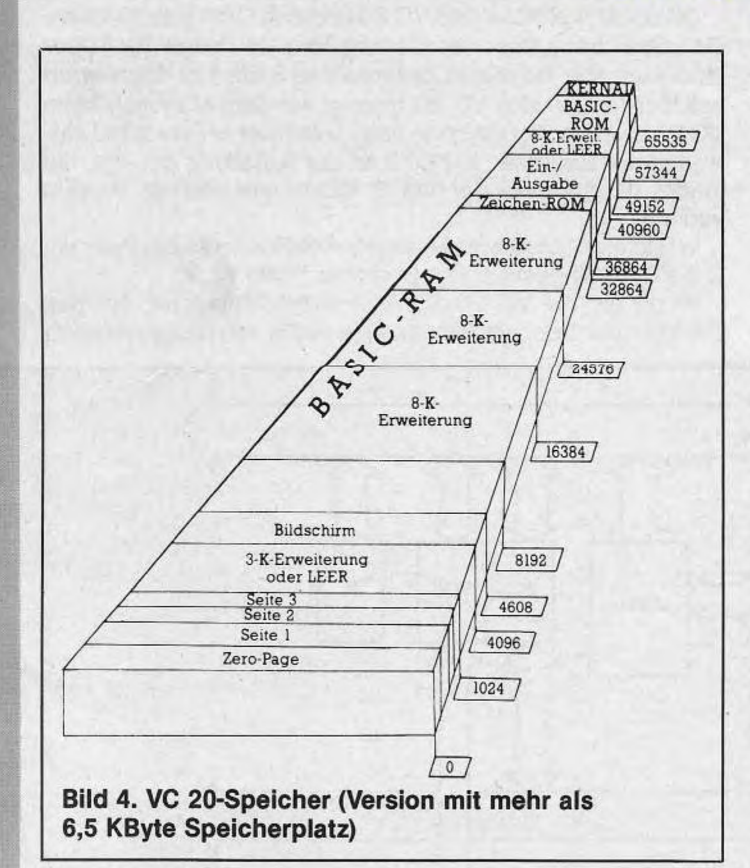


Bild 4. VC 20-Speicher (Version mit mehr als 6,5 KByte Speicherplatz)

wo manche Bereiche sogar zwei Etagen haben (soll heißen: mehrfach belegt sind). Durch die Eigenart des C 64 aber, im Normalfall das Basic-ROM, die Ein- und Ausgabebausteine und das Betriebssystem eingeschaltet zu haben, kann man ihn eigentlich genauso behandeln wie einen VC 20, bei dem die genannten ROM-Bausteine, - und zwar das Basic-ROM —, um 8 KByte verschoben sind. Die Unterschiede der ROM-Inhalte können fast vernachlässigt werden. Wir werden im Einzelfall darauf zu sprechen kommen. Bei den Ein- und Ausgabebausteinen liegen allerdings größere Unterschiede.

Die Seiten 0 bis 3 (eine Seite oder auch page enthält 256 Byte und man zählt oft auch in diesen Seiten, wenn vom Speicher die Rede ist), sind sich ebenfalls sehr ähnlich und die wenigen Unterschiede werden uns ebenfalls noch beschäftigen. Der Bildschirm liegt bei der Grundversion und der mit der 3-KByte-Erweiterung von 7680 bis 8191, in der Version mit mehr als 6,5 KByte von 4096 bis 4607 und beim C 64 von 1024 bis 2047. Der Bildschirmfarbspeicher liegt - bei gleicher Reihenfolge - von 37888 bis 38399, beziehungsweise von 38400 bis 38911 und schließlich von 55296 bis 56295. Der Basic-RAM-Bereich beginnt beim C 64 im Normalfall bei 2048 und endet bei 40959. Beim VC 20 ist das natürlich wieder von der jeweiligen Erweiterung abhängig (Tabelle 1).

Grundversion	:Basic-Start	4096	Basic-Ende	7679
+3-K-Erweiterung	: —"—	1024,	—"—	7679
+8-K-Erweiterung	: —"—	4608,	—"—	16383
+16-K-Erweiterung	: —"—	4608,	—"—	24575
+24-K-Erweiterung	: —"—	4608,	—"—	32767

Tabelle 1. Basic-Start und -Endadressen beim VC 20 mit verschiedenem Speicherausbau

Dies gilt - wie Sie leicht auch aus Bild 4 sehen können - auch dann, wenn zu den 8 KByte/16 KByte/24-KByte-Erweiterungen noch die 3-KByte-Erweiterung und die 'KByte-Erweiterung im hohen Speicherbereich (40960 bis 49151) verwendet werden. Diese letztgenannten Adressenbereiche sind dann gut als geschützte RAM-Bereiche für Maschinensprache zu verwenden, ebenso wie beim C 64 der Speicherabschnitt von 49152 bis 53247.

7. Auskunft über das Befinden unseres Computers: die Register-Anzeige

Bisher haben wir uns mit dem Innenleben unserer Computer auseinandergesetzt und die wichtigsten Teile der Hardware kennengelernt. Jetzt kommen wir zur Software, nämlich zum Assembler. Wenn Sie jetzt den SMON einschalten, meldet er sich mit einer Registeranzeige (Bild 5).

Die angezeigten Werte sind Beispiele, wie sie beim C 64 auftreten können. PC ist der Programmzähler, der immer auf den nächsten zu holenden Befehl zeigt. (Der Wert $E147 rührt vom SYS-Aufruf, mit dem ich meinen Assembler starte). IRQ zeigt uns an, auf welche Adresse der sogenannte Interrupt-Vektor gestellt ist. Das ist das Byte-Paar 788 (LSB) und 789 (MSB). Auf den Wert $EA31 zeigt es im Normalfall.

Die nächsten acht Angaben beziehen sich auf das Prozessorstatusregister, das wir zuletzt P genannt haben. Die Bedeutung der einzelnen »Flaggen« zeigt Ihnen Bild 6.

AC ist der aktuelle Inhalt des Akkus. XR zeigt an, was im X-Register und YR was im Y-Register enthalten ist. SP (von Stackpointer = Stapelzeiger) gibt uns Auskunft über den freien Platz im Stapelregister. Damit wissen wir genau, was in diesem Moment in unserem Computer vorgeht. So fremd Ihnen das alles im Augenblick noch vorkommt, bald werden Sie mit dieser Registeranzeige auf vertrautem Fuß stehen.

8. Wie sieht ein Assemblerprogramm aus?

Das menschliche Gehirn hat dem des Computers vieles voraus. Dazu gehört beispielsweise, daß ein Mensch allerlei Dinge gleichzeitig tun kann: gehen, sprechen, Musik hören, lächeln, Handbewegungen ausführen, womöglich dabei auch noch etwas kauen und so weiter. Ein Computer ist dazu nicht imstande. Er erledigt eine kleine Aufgabe nach der anderen. Weil er das so schnell macht, hat es für uns den Anschein, es geschähe alles gleichzeitig. Das Maschinenprogramm ist eine Kette solcher kleiner Aufgaben. Das erste Glied daraus, das wir kennenlernen wollen ist der Befehl

    LDA.

Das bedeutet: Lade den Akkumulator. Alle Assembler-Befehlsworte bestehen aus drei Buchstaben wie dieser hier auch. Wir haben in der ersten Folge schon gesagt, daß einem solchen Befehl eine 8-Bit-Codezahl entspricht. Das ist hier $A9 oder binär 1010 1001 oder schließlich dezimal 169. Die Codezahl muß in einem Speicherplatz stehen, zum Beispiel in $1500 (entspricht dez. 5376). Assemblerlistings sehen dann so aus:

    1500 LDA

Hier tritt also die Speicherplatznummer mit einem nachfolgenden Befehl anstelle der von Basic gewohnten Zeilennummer.

Es fehlt noch etwas Entscheidendes: Was soll denn in den Akku geladen werden? Genauso wie es in Basic Befehle gibt, die für sich alleine stehen können wie CLR oder LIST, gibt es auch im Assembler solche Befehle. Weitaus häufiger sind aber hier Befehle, die ein Argument erfordern (in Basic zum Beispiel PEEK(100). Dabei ist 100 das Argument). In Assembler gibt es zwei Sorten von Argumenten. Solche, die in einem Speicherplatz unterzubringen sind und andere, die zwei Byte brauchen. Mit dem Befehlswort (hier also LDA) zusammen, existieren in Assembler also 1-Byte-Befehle, 2-Byte-Befehle und 3-Byte-Befehle.

PC	IRQ	NV-BDIZC	AC	XR	YR	SP
E147	EA31	10110000	00	00	00	F8

Bild 5. Eine Registeranzeige

N	V	—	B	D	J	Z	C
Negativ-Flagge	Über-lauf-Flagge	unbe-nutzt	Abbruch-Flagge	Dezimal-Flagge	Interrupt-Flagge	Zero-(Null) Flagge	Carry-(Über-trag) Flagge

Bild 6. Das Prozessor-Status-Register P: die Flaggen

Befehls-wort	Adressierung	Byte-anzahl	Code HEX	Code DEZ	Dauer in Takt-zyklen	Beein-flussung von Flaggen
LDA	unmittelbar	2	A9	169	2	N, Z
	absolut	3	AD	173	4	N, Z
LDX	unmittelbar	2	A2	162	2	N, Z
	absolut	3	AE	174	4	N, Z
LDY	unmittelbar	2	A0	160	2	N, Z
	absolut	3	AC	172	4	N, Z
STA	absolut	3	8D	141	4	keine
STX	absolut	3	8E	142	4	keine
STY	absolut	3	8C	140	4	keine
RTS	implizit	1	60	96	6	keine

Bild 7. Die ersten sieben Befehle

Das Argument von LDA ist also das, was in den Akku soll. Laden wir deshalb mal eine 1 in den Akku:

    1500 LDA #$01

Wir haben jetzt einen 2-Byte-Befehl erzeugt. Was aber bedeuten »#« und »$« dabei? $ ist leicht zu erklären. Die große Mehrzahl der Assembler nimmt bei Zahlenangaben Hexadezimalzahlen an. Bei einigen muß man dies durch das $-Zeichen kennzeichnen. Manche Assembler lassen auch Binärzahlen, Dezimalzahlen und sogar ASCII-Zeichen als Argumente zu. Für jede Eingabeart steht dann vor dem Argument ein Zeichen, das die Art des Argumentes angibt, zum Beispiel häufig »!« für Dezimalzahlen oder »%« für Binärzahlen. Nun zum #-Zeichen. Es gibt viele Arten, den Akku zu laden. Direkt mit einer Zahl – wie wir hier —, aber zum Beispiel auch mit dem Inhalt eines anderen Speichers und so weiter. Man spricht von der sogenannten Adressierung.

Es gibt eine ganze Menge davon und jede wird auf eindeutige Weise gekennzeichnet. Wenn wir in unserem Akku eine Zahl laden, dann ist das die »unmittelbare« Adressierung und die kennzeichnet man mit dem #-Zeichen.

Wenn in Speicherstelle $1500 die Codezahl für LDA steht, dann muß die 1 in der Speicherstelle $1501 stehen, wie es sich für einen 2-Byte-Befehl gehört. Wenn Sie nun die Assemblerzeile eingegeben haben und ⟨RETURN⟩ drücken, dann taucht auf dem Bildschirm eine Fehlermeldung auf (bei vielen Assemblern). Wir müssen vorher nämlich noch unserem Software-Instrument sagen, jetzt zu assemblieren. Wie das geschieht, ist auch wieder von Assembler zu Assembler verschieden. Die meisten erwarten, daß man vor der Zeile noch ein A eingibt (zum Beispiel bei dem C 128):

    A 1500 LDA #$01

Wenn Sie jetzt ⟨RETURN⟩ drücken, zeigt der Bildschirm:

    A 1500 LDA #$01
    A 1502

und meistens einen blinkenden Cursor, der auf die nächste Eingabe wartet. $ 1502 ist die nächste freie Speicherstelle, und wenn beim Programmablauf der Programmzähler nach dem LDA #$01 auf $1502 deutet, dann erwartet er dort den nächsten Befehl. Wenn dort Unsinn steht, dann stürzt der Computer im allgemeinen ab, je nachdem, welcher Code dann hier zufällig enthalten ist. Wir haben ja 256 Möglichkeiten dafür: $00 bis $FF. Im Gegensatz zu Basic, wo man durch den Interpreter die Möglichkeit hat, Zeilennummern zu bauen wie man will, muß hier das Programm eine ununterbrochene Perlenschnur von Befehlen in Speicherstellen sein. Durch einige Befehle läßt sich dieses Prinzip allerdings durchbrechen.

Damit wir die Wirkung von Befehlen sehen können, greife ich auf einen Befehl vor, der ähnlich dem STOP in Basic einen Programmabbruch bewirkt: BRK. Die genaue Funktion soll erst später erklärt werden, aber wir sehen jedenfalls dann, wenn ein Maschinenprogramm auf einen BRK-Befehl läuft, die Registerinhalte angezeigt. Das ist in den meisten Assemblern eingebaut. Wir ergänzen jetzt:

    A 1502 BRK

Damit erstmal genug. Steigen Sie aus dem Assembler aus und starten Sie das Programm. In den meisten Assemblern geht das mit

    G 1500

oder sonst von Basic aus mit SYS 5376. Jetzt werden wieder die Register angezeigt. Der Programmzähler steht auf 1503, im Akku steht 01, alle Flaggen außer der Breakflagge sind Null (die unbenutzte Flagge steht immer auf 1). Jetzt ändern wir das Argument:

    A 1500 LDA #$00
    A 1502 BRK

Wir starten wieder und sehen uns die Register an: Programmzähler 1503, Akku jetzt 00, aber bei den Flaggen hat

sich etwas verändert: Die Zero-Flagge ist auf 1 gesetzt. Wir sehen also: Diese Flagge bleibt so lange ungesetzt, solange nicht eine Null im Akku auftaucht, erst dann wird sie 1.
Noch einmal ändern wir das Programm:

```
A 1500 LDA #$FF
A 1502 BRK
```

Nach erneutem Start steht das Erwartete in den Registern, nur bei den Flaggen ist etwas Merkwürdiges passiert: Die Vorzeichenflagge steht auf 1. Das bedeutet, im Akku soll eine negative Zahl stehen! Nun wissen wir aber, daß $FF = dez. 255 ist. Dieses Rätsel wird uns noch eine Weile begleiten. Es sei hier nur bemerkt, daß kein Fehler vorliegt: Immer wenn in einer Zahl das Bit 7 gleich 1 ist, geht die Vorzeichenflagge auf 1. Die Lösung des Rätsels werden wir bei den negativen Binärzahlen finden.

Wir schließen aus alledem: Der LDA-Befehl beeinflußt die Vorzeichen- und die Zeroflagge.

9. Die absolute Adressierung

STA heißt »STore Accumulator«, also »lege Akkuinhalt ab«. Wie Sie sich denken können, muß auch hier ein Argument auftauchen, nämlich wohin abgelegt werden soll. Wir legen unseren Akkuinhalt in die erste Bildschirmspeicherstelle (C 64:$0400, VC 20 Grundversion: $1E00, VC 20 mit Erweiterung: $1000). Unser Programm muß also so aussehen:

```
A 1500 LDA #$01
A 1502 STA $0400
oder die entsprechende Adresse
(siehe oben).
```

Mit diesem STA-Befehl lernen wir eine neue Adressierungsart kennen: die »absolute« Adressierung. Sie ist daran zu erkennen, daß kein besonderes Merkmal verwendet wird. Die Adresse $ 0400 ist nicht in einem Byte darstellbar, sondern wird aufgeteilt auf zwei Bytes. Im Speicher steht jetzt:

```
1500 LDA #
1501 $ 01
1502 STA
1503 $ 00      »das ist das LSB«
1504 $ 04      »das ist das MSB«
```

Hier liegt also ein 3-Byte-Befehl vor und die nächste freie Speicherstelle ist $ 1505.

Vom Basic her wissen Sie, daß 1 der Bildschirmcode für den Buchstaben A ist und daß man jeder Bildschirmspeicherstelle auch eine Bildschirmfarbspeicherstelle zuordnet. Um ein eingeschriebenes Zeichen vom Hintergrund abzuheben, muß man dort dann eine Farbinformation eingeben. Der Start dieses Bildschirmspeichers liegt so:

```
C 64:                $ D800
VC 20 (Grundv.):     $9400
VC 20 (Erw. Vers.):  $9600.
```

Der Farbe Schwarz entspricht die Codezahl 0. Wir ergänzen unser Programm durch:

```
A 1505 LDA #$00
```

A 1507 STA $D800 (oder entsprechender Speicher, siehe oben). Die nächste freie Adresse ist nun $150A. Unser Programm soll jetzt abgeschlossen sein. Damit der Computer aber beim Programmzählerstand $150A nicht Unsinn vorfindet, muß – ähnlich wie bei END in Basic – das Programm auf irgendeine Weise beendet werden. Das kann durch BRK geschehen. Wir wollen aber den dritten Assembler-Befehl kennenlernen:

```
RTS
```

Das heißt »Return From Subroutine«, also »Rückkehr aus Unterprogramm«. In unserem Fall bewirkt das eine Rückkehr zum Basic. Wie Sie sehen, ist das ein 1-Byte-Befehl, also ohne Argument. Auch hier spricht man von einer Adressierungsart, nämlich der »impliziten«-Adressierung. Man erkennt sie am Fehlen des Argumentes. Die Adresse ist implizit, das heißt im Befehl selbst enthalten. Dies ist nämlich ein Befehl, der immer an den Programmzähler gerichtet ist. Der Computer holt sich vom Stapel-Speicher die dort zuoberst liegende Adresse, das ist die, bei der der Computer in ein Unterprogramm gesprungen ist oder aber die, bei der der Computer Basic verlassen hat. Wir ergänzen also noch:

```
A 150A RTS
```

und starten das Programm, zum Beispiel von Basic aus mit SYS 5376. Natürlich taucht dann in der linken oberen Ecke des Bildschirmes ein schwarzes A auf. Hier noch der Basic-Lader:

```
10 FOR I=5376 TO 5386:READ A:POKE I,A:NEXT I:END
20 DATA 169,1,141,0,4*,169,0,141,0,216*,96.
```

Die mit * markierten Zahlen müssen für den VC 20 verändert werden: Grundversion: 30 und 148.
Erweiterung: 16 und 150.

10. Vier neue Befehle

Eine Kombination von LDA mit STA ist vergleichbar mit dem POKE-Befehl in Basic. Man kann in Assembler nicht direkt eine Zahl in einen Speicher einschreiben, sondern muß den Umweg über den Akku machen. Außer dem Akku eignen sich dazu aber auch das X-Register und das Y-Register. Hierfür gibt es die Befehle LDX (lade X-Register), STX (lege X-Register-Inhalt ab), LDY (lade Y-Register) und schließlich STY (lege Y-Register-Inhalt ab). Sie können das übungshalber an unserem kleinen Programm ausprobieren. An dem folgenden Programm sehen Sie noch eine Eigenart der drei Register (Akku, X-Register, Y-Register):

```
A 1500 LDA  #$01
A 1502 LDX  #$00
A 1504 LDY  #$02
A 1506 STA  $0400
A 1509 STX  $D800
A 150C STY  $0401
A 150F STX  $D801
A 1512 STA  $0402
A 1515 STX  $D802
A 1518 RTS
```

Für den VC 20 werden die entsprechenden Speicherstellen für Bildschirm- und Bildschirmfarbspeicher eingesetzt. Dieses Programm druckt – wie erwartet – »ABA« in die linke obere Ecke des Bildschirms. Dabei ist das X-Register dreimal ausgelesen worden und der Akku zweimal. Sie sehen also, daß die Registerinhalte durch die STA-, STX-, STY-Befehle nicht verändert werden.

Wir wollen noch etwas ausprobieren. Bisher haben wir den LDA-Befehl nur mit der »unmittelbaren« Adressierung kennengelernt. LDA, LDX, LDY können auch »absolut« adressiert werden.

```
A 1518 LDA $D800
```

Damit laden wir den Inhalt der Speicherstelle $ D800 (beim VC 20 die anderen Adressen des Bildschirmfarbspeichers) in den Akku. Der Inhalt ist seit $1509 eine Null. Jetzt weiter:

```
A 151B STA $0403
A 151E STX $D803
A 1521 RTS
```

Das müßte beim Ablauf des Programms noch einen Klammeraffen (@mit Bildschirmcode 0) an die vierte Stelle plazieren, was Sie durch SYS 5376 leicht nachprüfen können. Sie sehen, daß man mit diesen sieben Befehlen schon eine Menge anfangen kann.

Wir kommen noch einmal zur Adressierung. Ich hatte Ihnen gesagt, daß LDA #$01 ein 2-Byte-Befehl mit unmittelbarer

Adressierung ist (ein Byte für LDA und eines für 01), LDA $D800 ist ein 3-Byte-Befehl (ein Byte für LDA, je eines für das LSB und das MSB von $D800) mit absoluter Adressierung. Da werden Sie sich doch sicher schon gefragt haben, wo die Adressierung bleibt! Wenn aber kein Byte für die Adressenmarkierung (zum Beispiel #) reserviert ist, muß die Kennzeichnung irgendwie anders sein. Wenn Sie einen Disassembler zur Verfügung haben, dann sehen Sie sich damit unser Programm an. Fast jeder Disassembler gibt neben dem Assemblertext auch Byte für Byte in Hexadezimalzahlen die Codes an. Wenn Sie nun die beiden Befehle LDA #$01 und LDA $d800 von den Codes her untersuchen, sehen Sie folgendes:

```
1500 A9 01        LDA #$01
```

und

```
1518 AD 00 D8  LDA $D800
```

Offensichtlich gehört jeweils das erste angezeigte Byte zu LDA. Sie sind aber verschieden! Wir sehen daraus, daß die Codezahl für einen Befehl gleich zwei Informationen enthält: das Befehlswort selbst (LDA) und die Adressierungsart.

Genauso wie man LDA sowohl unmittelbar als auch absolut ausführen kann, ist das auch mit LDX und LDY möglich. Bei den Befehlen STA, STX, STY ist eine unmittelbare Adressierung sinnlos. Für RTS kennt man nur eine implizite Adressierung. Wir fassen das alles in Bild 7 zusammen.

In den letzten Spalten von Bild 7 ist noch angegeben, inwieweit durch diese Befehle das Prozessorstatusregister beeinflußt wird, so wie wir es für den Befehl LDA schon ausprobiert haben. In der vorletzten Spalte sehen Sie, wie lange die Ausführung eines Befehls dauert. Wenn sie für einen Taktzyklus etwa eine Mikrosekunde rechnen, dann müßten Sie jetzt ausrechnen können, wie lange unser letztes Programm zur Bearbeitung braucht: 48 Mikrosekunden. Ein vergleichbares Basic-Programm braucht dazu etwa hundertmal so lange: zirka 0,05 Sekunden.

11. Die Zahlen der Assembler-Alchimisten

Ein bißchen von Assembler-Alchimie verstehen Sie jetzt schon mit diesen sieben Befehlen. Wir wollen uns nun die Zahlen ansehen, die hier Verwendung finden: das Binärsystem und das Hexadezimalsystem.

Die einzigen Ziffern, die unser Computer kennt, sind 0 und 1. Sie stehen für »Strom an« oder »Strom aus«, oder für »keine magnetische Erregung« oder »magnetische Erregung«. Deshalb ist es für uns als angehende Assembler-Alchimisten von großer Bedeutung – wir arbeiten ja ganz eng an der Hardware – dieses binäre Zahlensystem handhaben zu können. Das Hexadezimalsystem kennt der Computer eigentlich gar nicht. Wir verwenden es deswegen, weil es in einem besonders engen Zusammenhang mit Binärzahlen und dem Aufbau unseres Computers steht: Die größte einstellige Hex-Zahl ist $F, das entspricht genau 1111 im Binärsystem, also dem maximalen Füllungsgrad eines halben Bytes, das Nibble genannt wird. Ein ganzes Byte kann maximal $FF enthalten (binär 1111 1111) und der gesamte Speicheradressenbereich unseres Computers geht bis $FFFF (dezimal 65535). Eine einstellige Hex-Zahl paßt also in ein Nibble, eine zweistellige in ein Byte und eine dreistellige oder vierstellige in zwei Byte, weshalb man solche Hex-Adressen auch recht leicht in das LSB und das MSB (auch Low- und High-Byte genannt) aufteilen kann:

```
$  D8   00
   MSB LSB
```

Rechnen werden wir mit Hexadezimalzahlen nicht, dazu benutzen wir dann das Dezimalsystem oder – wenn es sich um computerinterne Vorgänge handelt – das Binärsystem.

Das Rechnen mit Binärzahlen funktioniert genauso wie das mit Dezimalzahlen. Es gilt also

0+0=0
0+1=1
1+0=1
1+1=10

wobei binär 10 gleich dezimal 2 ist. Als Beispiel können wir mal 2+1=3 im Binärsystem rechnen:

```
  10 entspricht dez. 2
+01 entspricht dez. 1
-----------------------
  11, was ja dezimal 3 ergibt.
```

Die Addition erfolgt also spaltenweise wie beim gewohnten dezimalen Addieren. Auch mit dem Übertrag läuft es wie im dezimalen. Beispiel: 2+2=4:

```
  10 entspricht dez. 2
+10 entspricht dez. 2
-----------------------
 100, was dezimal eine 4 ergibt.
```

In der zweiten Spalte wurde nach der Regel verfahren: 1+1=10. Rechnen wir noch 3+3=6:

```
  11 entspricht dez. 3
+11 entspricht dez. 3
-----------------------
 110, was dezimal eine 6 ergibt.
```

In der ersten Spalte wurde gerechnet 1+1=10, wobei nach dem alten Motto: 0 hin, 1 im Sinn die 0 unter den Strich gesetzt wurde. In der zweiten Spalte wird dann so verfahren: 1+1+1 (das ist die 1, die wir »im Sinn« hatten) = 11. Ich meine, daß Sie ohne Probleme die folgenden Übungsaufgaben lösen und dann jeweils dezimal das Ergebnis nachprüfen können: 10+5, 7+1, 16+16, 240+16, 62+65.

12. Eine Zauberformel der Assembler-Alchimisten: INX, INY, INC, DEX, DEY, DEC?

Wir wissen ja schon, daß man diese »Zauberformeln« entzaubern kann. INX heißt einfach »INCrement X-Register«, also Inhalt des X-Registers um 1 erhöhen. Es wird Ihnen sicher einleuchten, daß INY dasselbe mit dem Y-Register tut. Etwas weniger deutlich ist das bei INC. Das bedeutet »INCrement memory«, also zähle zum Inhalt einer Speicherstelle eins dazu. INX und INY enthalten alles, was dem Computer zu sagen ist, sind also offensichtlich 1-Byte-Befehle mit der in der letzten Folge schon kennengelernten impliziten Adressierung. Bei INC muß dem Computer noch gesagt werden, welche Speicherstelle er um 1 erhöhen soll. Es gehört also noch eine Adresse dazu. Das läßt diesen Befehl im allgemeinen zu einem 3-Byte-Befehl werden.

Das Umgekehrte leisten die Befehle DEX, DEY und DEC. Sie bedeuten nämlich »DECrement X-Register«, also »zähle das X-Register um eins herunter«, beziehungsweise das Y-Register oder – bei DEC – die angegebene Speicherstelle. Für die Adressierungsart und die Anzahl Bytes pro Befehl gilt hier das gleiche wie für die INX...-Befehle. Sehen wir uns das an einem kleinen Beispiel an:

```
1500    LDA #00
1502    LDX #01
1504    STA D800
1507    STX 0400
150A    INX
150B    STA D801
150E    STX 0401
1511    DEX
1512    STA D802
1515    STX 0402
1518    BRK
```

Wenn Sie das kleine Programm mit G 1500 starten, dann sollten Sie in der linken oberen Ecke des Bildschirms ABA in schwarzer Schrift stehen haben. Was ist geschehen? Wir haben den Inhalt des Akkus (=0, also Farbcode für schwarz) in das Bildschirm-Farbregister geschrieben (#D800), dann den Inhalt des X-Registers (1 = POKE-Code für den Buchstaben A) in die erste Bildschirm-Speicherzelle (#0400). Anschließend wurde das X-Register um 1 erhöht (2 = POKE-Code für den Buchstaben B) und dieser Inhalt in die zweite Bildschirmzelle geschrieben. Außerdem mußte natürlich auch dieser Bildschirm-Farbspeicherplatz mit dem Farbcode 0 belegt werden. Durch DEX wurde das X-Register wieder heruntergezählt, somit wieder ein A erzeugt und in die dritte Bildschirmstelle gedruckt.

Sie haben sicher schon bemerkt, daß man auf diese Weise Abläufe mitzählen kann. Soll zum Beispiel ein Vorgang 20 mal wiederholt werden, dann packt man ins X-Register (oder ins Y-Register oder in eine andere Speicherstelle) den Anfangswert 0, läßt den Computer eine Arbeit ausführen, erhöht das entsprechende Register oder die Speicherzelle um 1 mit INX, INY oder INC, prüft dann, ob dieser Inhalt schon 20 geworden ist und so weiter. Wie man diese Prüfung vornimmt, dazu kommen wir erst später bei den BRANCH-Befehlen. Das ist also ähnlich wie in Basic bei den FOR...NEXT-Schleifen: Dort wird eine Variable als Zähler verwendet, hier ein Register (oder eine Speicherstelle). Ebenso wie in Basic bei diesen Schleifen kann man auch hier rückwärts zählen mit DEX, DEY oder DEC. Das hat oft gewisse Vorzüge, was uns aber noch nicht kümmern soll.

Wenn wir diese Befehle als Zähler verwenden, sollten wir im Auge behalten, daß eine Speicherstelle (auch ein X- oder Y-Register) Zahlen nur von 0 bis 255 enthalten kann. Die höchste 8-Bit-Zahl ist ja:

```
dez. 255 = bin. 1111 1111
        +1              1
               ----------
ergibt:        (1) 0000 0000
```

Wenn wir also über 255 hinauszählen, ergibt sich wieder 0 und so weiter, weil ein Überlauf stattgefunden hat. Das 9.Bit paßt nicht mehr in das Byte hinein. Um nochmal genau sehen zu können, was unser Computer da tut, probieren Sie einmal aus:

```
1500 LDA #01
1502 BRK
```

Das soll uns die Register zunächst mal im Ausgangszustand zeigen. Nach G 1500 werden sie angezeigt:

```
AC XR YR N V-BDI ZC
01 00 00 0 0110000
```

Im Akku steht jetzt die dort eingeladene 1. Nun wollen wir das X-Register laden mit 255 (also $FF). Dazu ändern wir das Programm:

```
1502 LDX #FF
1504 BRK
```

Nach erneutem G 1500 zeigen die Register:

```
AC XR YR N V-BDI ZC
01 FF 00 1 0110000
```

Im X-Register steht nun die Zahl $FF. Bei den Flaggen hat sich die N-Flagge (die negative Zahlen anzeigen soll) auf 1 geschaltet!

Nun wollen wir das X-Register über 255 hinauszählen. Wir verändern das Programm nochmal:

```
1504 INX
1505 BRK
```

Der Start mit G 1500 liefert uns die folgende Registeranzeige:

```
AC XR YR N V-BDI ZC
01 00 00 0 0110010
```

Wie erwartet, ist der Überlauf des X-Registers eingetreten: Es ist jetzt Null. Die N-Flagge hat ihren gewohnten Wert 0 wieder angenommen und die Z-Flagge, die uns anzeigt, ob die letzte Operation eine Null erzeugt hat, ist jetzt gesetzt. Bei weiterem Hochzählen verschwindet die Z-Flagge wieder:

```
1505 INX
1506 BRK
```

G 1500 liefert den Registerinhalt:

```
AC XR YR N V-BDI ZC
01 01 00 0 0110000
```

Das gleiche passiert bei Verwendung des Y-Registers als Zähler, wie Sie leicht durch Austauschen aller auf X bezogenen Befehle feststellen können. Sehr nett ist es, diesen Befehlsablauf einmal für den INC-Befehl auf die Speicherstelle $0400 (Bildschirmspeicher links oben) bezogen ablaufen zu lassen. Wenn man darauf achtet, daß kein Hochscrollen des Bildschirms eintritt, kann man das Ergebnis außer in den Registern auch noch als Zeichen auf dem Bildschirm verfolgen. Der Beginn der Befehlsequenz ist dann sinnvollerweise:

```
1500 LDA #FF
1502 STA 0400
1505 BRK
```

Im folgenden setzt man dann anstelle von INX immer INC 0400 ein.

Was passiert beim Herunterzählen unter Null? Sie können das mit der gezeigten Befehlskette leicht verfolgen, indem Sie immer statt INX jetzt DEX setzen und die Register nicht mit $FF, sondern mit 01 laden. Es zeigt sich, daß beim Herabzählen nach der Null wieder 255 (=$FF) im Register zu finden ist. Die Reaktion der N- und der Z-Flagge auf den jeweiligen Registerinhalt ist die gleiche wie beim Hochzählen.

Es ist uns nun deutlich, daß diese sechs Befehle die N-Flagge und die Z-Flagge beeinflussen können. Diese Tatsache wird später noch eine große Rolle spielen, wenn es um die bereits erwähnte Schleifenkontrolle geht.

13. Noch ein alchimistischer Zahlentrick: BCD

Die Assembler-Alchimisten haben noch viel mehr Arten der Zahlen- und Zeichendarstellung auf Lager. Eine davon ist die Codierung als BCD-Zahlen. BCD kommt vom englischen »binary coded dezimal«, was bedeutet: Binär codierte Dezimalzahlen.

Zwischendurch möchte ich noch eine Bemerkung loswerden, die Sie als Trost auffassen sollen: Auch wenn wir später andere Zahlendarstellungen kennenlernen werden, es wird nicht so schwierig! Sogar so komplette Idioten wie Computer verstehen das, obwohl man ihnen alles haarklein vorkauen muß.

Wenden wir uns nun wieder den einfachen BCD-Zahlen zu. Alle Zahlen von 0 bis 9 lassen sich binär mit nur 4 Bit ausdrücken:

Binär	Dezimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9

Die weiteren Werte 1010 bis 1111 werden in der BCD-Codierung nicht benutzt. Liegt nun eine Dezimalzahl (zum Beispiel 12) vor, dann wird jede Stelle dieser Zahl (also die 1

und die 2) getrennt binär codiert. In unserem Beispiel mit der 12 wäre das dann 0001 für die 1 und 0010 für die 2. Somit ist die 12 im BCD-Code 0001 0010. Jede Ziffer erhält so ihr Nibble. Eine Zahl im BCD-Format hat deswegen keine feste Anzahl von Bytes, sondern die Byte-Zahl hängt von der Anzahl der Stellen ab. Die Zahl 1984 beispielsweise braucht 2 Byte: 0001 1001 1000 0100.

Schwierig gestaltet sich das Rechnen mit diesen Zahlen wegen der sechs unbenutzten Codes. Aber auch da habe ich einen Trost für Sie: Wir werden damit nicht rechnen. Wozu das ganze dann, werden Sie sich fragen? Der Grund für das alles ist, daß BCD-Zahlen im Gegensatz zu den Zahlen mit festem Format (die sonst verwendet werden) so eingegeben und verarbeitet werden können, wie sie vorliegen. Das ist im kaufmännischen Bereich manchmal notwendig, wo eben 1000 mal 0,1 Pfennige 1 Mark ergeben und Fehler unzulässig sind. Sollten Sie also vor dem Problem stehen, mit BCD-Zahlen rechnen zu müssen, grämen Sie sich nicht: Unser Prozessor kennt den Dezimalmodus. Er ist dann eingeschaltet, wenn die Dezimal-Flagge auf 1 gesetzt ist.

Damit sollen Sie dann auch noch gleich zwei neue Befehle kennenlernen: SED und CLD. Der erstere hat nichts mit Parteien zu tun, sondern ist die Abkürzung für »SEt Dezimalflag«, also setze die Dezimalflagge. So schalten Sie den Dezimal-Modus ein. Wie Sie sicher schon messerscharf geschlossen haben, heißt CLD »CLear Dezimal-flag«, also setze die Dezimalflagge auf Null, wodurch dieser Modus wieder auszuschalten ist.

Wichtig! Wenn Sie argwöhnen, daß in einem Programm irgendwann mal die Dezimal-Flagge gesetzt sein könnte, dann gehen Sie auf Nummer sicher und schieben Sie vor eine Rechenoperation, die nicht im Dezimalmodus laufen soll, ein CLD.

Beide Befehle sind 1-Byte-Befehle mit implizierter Adressierung. Sie beeinflussen lediglich die Dezimalflagge.

Wie schon mal betont: Der Computer ist strohdumm. Er kann nicht einmal auf normale Weise voneinander abziehen! Deswegen geht er den komplizierten Weg: Er addiert eine negative Zahl. Nur: Wie sehen negative Binärzahlen aus? Wir werden diese Frage in drei Etappen beantworten.

a) Man könnte eine Flagge setzen, die 1 ist bei negativen und 0 bei positiven Zahlen. Bei einigen Fließkommazahlen wird das auch so gemacht. Hier aber setzt man die Flagge direkt in die Zahl ein: Bit 7 jeder Zahl ist jetzt ein Vorzeichenmerkmal. Wenn dieses Bit 0 ist, handelt es sich um eine positive, wenn es 1 ist, um eine negative Zahl. Auf diese Weise ist also +1 wie bisher 0000 0001, wohingegen —1 jetzt 1000 0001 hieße. Damit wird allerdings der Zahlenbereich, der durch ein Byte auszudrücken ist, verschoben. 255=binär 1111 1111 kann so nicht mehr verwendet werden. Die größte Zahl, die jetzt ausgedrückt werden kann, ist 0111 1111 = dezimal 127. Die kleinste Zahl ist dann 1111 1111 = —127. Probieren wir mal aus, wie sich damit rechnen läßt:

```
+10      0000 1010
—6       1000 0110
--------------------
ergibt   1001 0000 = —16,
```

was offensichtlich falsch ist, denn nach Adam Riese sollte +4 herauskommen. So kann man also nicht rechnen!

Man nennt diese Art der Zahlendarstellung übrigens »signed binary«-Format, also in Deutsch: markierte Binärzahlen.

b) Der nächste Schritt ist das sogenannte Einerkomplement. Dabei tritt für die positiven Zahlen keine Änderung ein. Die negativen entstehen aus den positiven durch Komplementbildung, das heißt jedes Bit der positiven Zahl wird in sein Gegenteil verkehrt, wie es das folgende Beispiel zeigen soll:

```
  0000 1100  ist +12,
dann ist das Einerkomplement:
  1111 0011  = —12.
```

Interessanterweise taucht hier auch wieder das Merkmal der »signed binary«-Zahlen auf: die 1 in Bit 7 bei negativen Zahlen. Beschränkt man sich auf den Zahlenbereich, der für die »signed binary«-Zahlen gültig war, dann hätten wir jetzt beide Darstellungsweisen miteinander vereint. Nun müssen wir natürlich noch feststellen, ob man so auch rechnen kann.

```
+8           0000 1000
—6           1111 1001
in Einerkomplementdarstellung
----------------------
ergibt   (1) 0000 0001
```

was 1 mit einem Übertrag ergäbe, jedenfalls nicht 2, wie es sich gehört. Also ist auch die Einerkomplementdarstellung noch nicht das Gelbe vom Ei.

c) Ich will Sie nicht länger auf die Folter spannen: Wenn man zum Einerkomplement einer Zahl noch 1 dazuzählt, erhält man das Zweierkomplement. Und genauso werden negative Zahlen in unserem Computer gehandhabt. Die positiven Zahlen bleiben unverändert. Von den negativen bildet man das Zweierkomplement wie zum Beispiel hier mit der Zahl —12:

```
12    0000 1100   normale Binärdarstellung
—12   1111 0011   Einerkomplement
+1    0000 0001   addieren
----------------
—12   1111 0100   Zweierkomplement
```

Jetzt wollen wir auch diese Zahlenart ausgiebig testen: Wir rechnen nochmal 8-6:

```
+8    0000 1000
—6    1111 1010   das ist —6 in der
                  Zweierkomplementdarstellung.
----------------
ergibt
  (1) 0000 0010
```

also 2 mit einem Übertrag, der ignoriert wird. Das Ergebnis ist richtig. Wenn bei einer solchen Rechnung eine negative Zahl herauskommt, ist sie nicht leicht zu erkennen. In solchen Fällen kehrt man das Vorzeichen um, indem man das Zweierkomplement berechnet. Das machen wir mal am Beispiel 5—6:

```
+5    0000 0101
—6    1111 1010   das ist wieder unser Zweier-
                  komplement von 6, also —6
----------------
ergibt1111 1111
```

das ist —1 in der Zweierkomplementdarstellung. Zur Kontrolle nun die Vorzeichenumkehr durch Umrechnen ins Zweierkomplement:

```
Einerkomplement davon   0000 0000
plus 1                  0000 0001
---------------------------------
ergibt                  0000 0001
```

also wie erwartet +1.

Auf diese Weise rechnet unser Computer mit negativen Zahlen. Negative ganze Zahlen speichert er im Zweierkomplement-Format. Auch wenn wir nun etwas vorgreifen müssen, wollen wir uns das ansehen. Dazu schalten Sie am besten erst einmal den Computer aus und laden dann den SMON beziehungsweise ihren Assembler. Dann bauen wir ein kleines Basic-Programm:

```
10 A%=—12
20 END
```

14. Wie Variable im Speicher stehen

Noch nicht RUN eingeben! Zuerst schalten Sie den Maschinensprachmonitor ein und wir sehen uns das Programm so an, wie es im Speicher steht. Der Basic-Speicher des C 64


Ungla


blich!
und MULTIPLAN

beginnt im Normalfall bei $0800. Wir geben also den Monitorbefehl M 0800.

Uns genügen schon die Speicherplätze bis $081C. Nun sehen wir das nackte Basic-Programm im Speicher, so wie es uns C. Sauer in seinem Artikel »Der gläserne VC 20, Teil 1« im 64'er, Ausgabe 9/84 auf Seite 156 beschrieben hat.

In Bild 8 ist unser Speicherinhalt kommentiert zu sehen. Das Programm endet im Speicherplatz $0813. Das Kennzeichen für Programmende sind zwei aufeinanderfolgende Bytes mit dem Wort Null. Dahinter werden die Variablen abgelegt, sobald das Programm gestartet wird. Wir steigen aus dem Monitor durch X aus und starten das Programm mit RUN. Jetzt sehen wir nochmal in den Speicher. Bis $0813 hat sich nichts verändert. Danach aber ist jetzt in 7 Bytes die Variable A% abgelegt. Das zeigt Bild 9.

Zunächst einmal die Bytes $0814 und $0815: Hier wird der Variablenname und -typ angegeben. Der Typ ist aus den Bits 7 zu erkennen. Sind beide (wie hier) gleich 1, dann handelt es sich um eine Integervariable (also eine ganze Zahl). Läßt man die Kennbits außer acht, zeigt sich, daß in $0814 der Code für den Buchstaben A steht und $0815 nur den Wert 0 enthält. Nun zum Rest: Der C 64 legt Integers in nur 2 Byte ab – die restlichen 3 Byte $0818 bis $081A bleiben unbenutzt. Das ist auch dann der Fall, wenn danach noch weitere Variable kommen. Es bringt also keine Speicherersparnis (VC 20-Benutzer aufgepaßt!), wenn man mit Ganzzahlvariablen arbeitet!

In $0817 steht $F4, welches binär ausgedrückt 1111 0100 ist. Das kennen wir noch von weiter oben als die —12 im Zweierkomplement-Format. Woher kommt $FF in Speicherzelle $0816? Wie gesagt, die Integers werden in 2 Byte gespeichert, und wenn wir —12 in 16 Bit ausdrücken, dann sieht das so aus:

+12	0000 0000 0000 1100
Einerkomplement:	1111 1111 1111 0011
plus 1	0000 0000 0000 0001
ergibt —12:	1111 1111 1111 0100
	MSB =$FF / LSB =$F4

als 16-Bit-Zweierkomplement.

Die größte positive ganze Zahl, die man in 2 Byte ausdrücken kann, ist 32767, was binär

0111 1111 1111 1111

ergibt. Die kleinste ist

1000 0000 0000 0000

also —32768. Das ist der Grund dafür, daß der C 64 Integers größer als 32767 oder kleiner als —32767 dankend mit ILLEGAL QUANTITY ERROR ablehnt, wenn sie als Argument verwendet werden. (Die Zahl —32768 kann als Ergebnis von logischen Operationen auftauchen.)

Damit will ich Sie erstmal von den Zahlenspielereien erlösen. Sie können die Art des Abziehens von Zahlen durch Addieren des Zweierkomplementes bis zum nächsten Mal an weiteren Beispielen üben. Wenn Sie das mit 16-Bit-Zahlen tun, werden Sie bald feststellen, daß noch nicht alles so funktioniert wie es sollte...

Wir können jetzt übrigens auch das Rätsel lösen, weshalb bei positiven Zahlen (zum Beispiel LDA #FF) die Negativ-Flagge auf 1 geht: Die Flagge wird immer dann gezückt, wenn eine Zahl auftritt, die in Bit 7 eine 1 aufweist. Ganz einfach, gell?

15. Ein wirkungsvolles Zweiglein: BNE

Vermutlich raucht Ihnen nach soviel Zahlensalat der Kopf. Deshalb sollen Sie zur Entspannung noch einen neuen Assembler-Befehl kennenlernen und auch gleich ein nützliches Programmbeispiel dazu.

BNE heißt »Branch if Not Equal zero«, was man übersetzen kann mit »verzweige, wenn ungleich Null«. Genauer gesagt: Es wird dann verzweigt – also zu einer angegebenen Adresse gesprungen —, wenn die Z-Flagge (die haben wir bei den INX,DEX...-Befehlen genauer untersucht) nicht gesetzt ist, also 0 zeigt. Sehen wir uns das mal an der nachfolgenden Verzögerungsschleife an, deren Flußdiagramm Bild 10 zeigt.

Das Programm dazu:

```
1500    LDX #FF
1502    LDY #FF
1504    DEY
1505    BNE 1504
1507    DEX
1508    BNE 1502
150A    BRK
```


Zunächst einmal werden das X- und das Y-Register als Zähler initialisiert (also mit einem Ausgangswert geladen). Mit dem vorhin behandelten Befehl DEY wird dann das Y-Register um 1 heruntergezählt, was jetzt $FE ergibt. Für die Nullflagge (Z) bedeutet das den Inhalt 0, denn es liegt kein Grund vor, sie zu setzen (also eine 1 dort anzuzeigen), weil noch keine Null aufgetreten ist. Bei der nachfolgenden Prüfung durch BNE wird also eine Verzweigung nach 1504 das Ergebnis sein, worauf das Y-Register weiter verringert und dann die Z-Flagge erneut geprüft wird und so weiter. Das geht so lange, bis nun wirklich endlich die Null im Y-Register erreicht ist. In diesem Fall zählt DEX nun das X-Register herunter und der nächste BNE-Befehl führt zum Sprung nach 1502, wo das Y-Register wieder auf $FF gesetzt wird. Auf diese Weise wird die äußere Schleife 255mal und die innere 65025mal durchlaufen.

0800	**00**	**0C**	**08**	**0A**	**00**	**41**	**25**	**B2**
		080C Koppeladresse		000A Zeilennr.10		A	%	= Token
0808	**AB**	**31**	**32**	**00**	**12**	**08**	**14**	**00**
	— Token	1	2	Zeilen-ende	0812 Koppeladresse		0014 Zeilennr.20	
0810	**80**	**00**	**00**	**00**	**FF**	**FF**	**FF**	**FF**
	END Token	Zeilen-ende	Programm-ende		Leerer Speicher			

Bild 8. Der Monitor zeigt das nackte Programm im Speicher

Speicherstelle	0814	0815	0816	0817	0818 bis 081A
Byte	1	2	3	4	5 — 7
Inhalt	C1 1100 0001	80 1000 0000	FF 1111 1111	F4 1111 0100	00 00 00
	Kennbits 7 für Integer 0100 0001 0000 0000 ≙ 65 Code für A		MSB	LSB von —12	unbenutzt bei Integerzahlen
	Variablenname und -typ		Variablenwert		

Bild 9. So werden Integer-Variable aus Basic-Programmen vom C 64 im Speicher eingerichtet

Sie haben beim Eingeben des Programmes vermutlich etwas gestutzt, als der Assembler nach dem BNE 1504 als nächste Adresse statt dem erwarteten 1508 eine 1507 ausgegeben hat. Der Befehl sieht zwar wie ein 3-Byte-Befehl aus, ist aber nur ein 2-Byte-Befehl! Das liegt an der speziellen Art der Adressierung von solchen Branch-Anweisungen: Der sogenannten relativen Adressierung, die wir aber erst später mit den anderen Branch-Befehlen behandeln werden.

Wenn Sie das Programm mit G 1500 starten, werden Sie - obwohl alles in Maschinensprache schnell läuft - eine merkliche Verzögerung feststellen, bevor die Registeranzeige auftaucht. Noch längere Verzögerungen lassen sich ohne weiteres erreichen, indem man mehr Schleifen ineinanderschachtelt. Dabei verwendet man dann den DEC-Befehl.

In der Tabelle 2 sind auch die Zyklen angegeben, die die neu gelernten Befehle zur Abarbeitung benötigen. Mit solchen Angaben lassen sich recht genau definierte Zeiten einstellen, in denen der Computer nichts anderes tut als durch das Programm zu flitzen. Wozu das dient, braucht wohl kaum noch gesagt werden: Wenn Sie zum Beispiel einen Text auf dem Bildschirm lesen wollen, bevor das Programm weiterläuft oder wenn Sie mit Peripherie arbeiten, die langsamer als das Programm ist oder... Allerdings muß noch gesagt werden, daß es noch elegantere Methoden zur Verzögerungs-Programmierung gibt als das Lahmlegen des Computers, aber dazu kommen wir erst später.

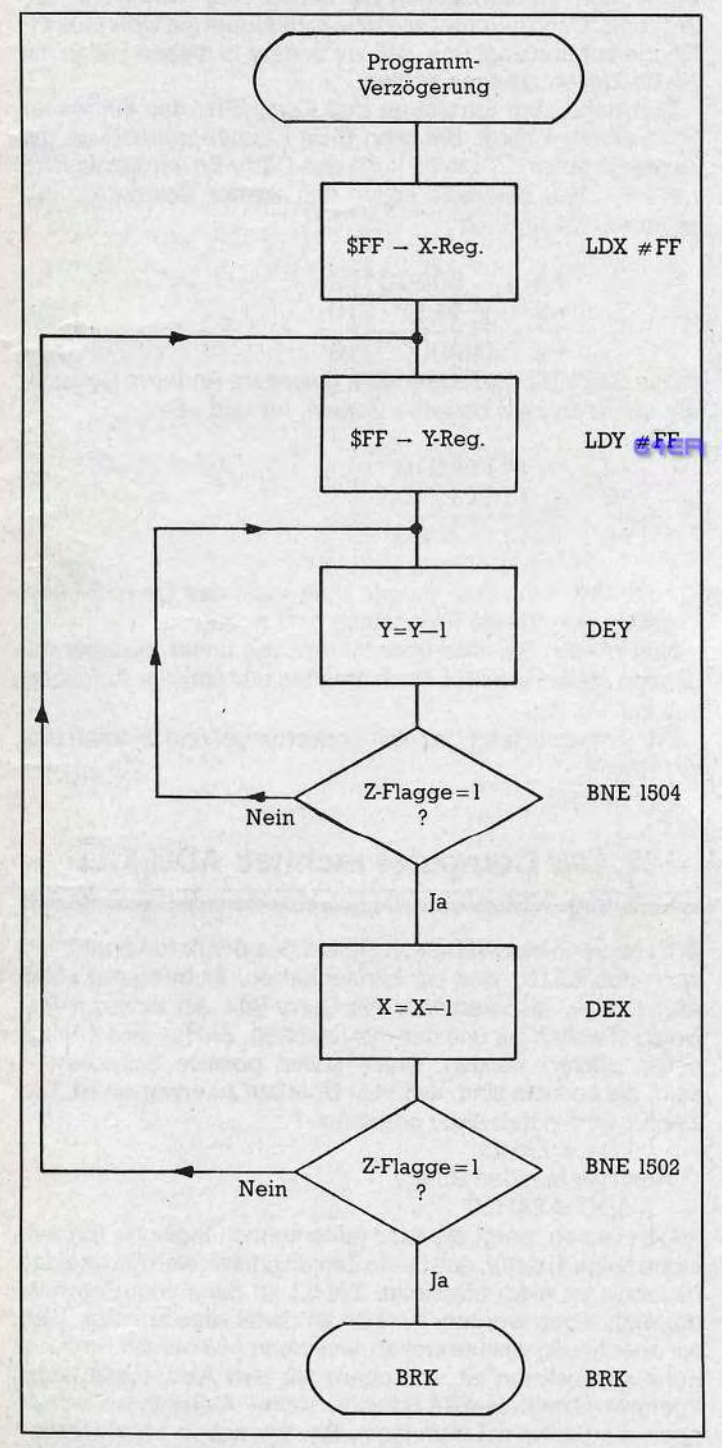


Bild 10. Flußdiagramm zur Verzögerungsschleife

16. Herr Carry und der V-Mann

Neun neue Befehle haben wir bisher kennengelernt und wir wissen nun, wie unser Computer ganze Zahlen (sogenannte Integers) abspeichert. Zur Erinnerung: Das geschieht im Zweierkomplement-Format. Das Bit 7 einer 8-Bit-Zahl dient dabei als Vorzeichen-Merkmal: Wenn es 0 ist, liegt eine positive Zahl vor, die genauso aussieht, wie wir bislang immer Binärzahlen kannten. Ist das Bit 7 aber eine 1, dann haben wir es mit einer negativen Zahl in der Zweierkomplement-Darstellung zu tun. Wenn wir - wie unser Computer - zur Verarbeitung ganzer Zahlen 16 Bits (also 2 Bytes) verwenden, dann ist eben Bit 15 anstelle von Bit 7 das Vorzeichenbit.

Wenn Sie ein bißchen mit solchen Zahlen gerechnet haben, konnten Sie sicher feststellen, daß zwar oft das richtige Ergebnis herauskam - aber leider nicht immer.

Keine Angst, wir sind nicht ins Krimi- oder Agentenmilieu gewechselt! Wir haben es mit zwei Flaggen zu tun, der Carry- und der V-Flagge. »To carry« heißt auf deutsch etwa »tragen«. In der Registeranzeige ist diese Flagge immer mit C gekennzeichnet. Was wird denn hier getragen? Das ergründen wir am besten an einem Beispiel. Dazu rechnen wir mit normalen Binärzahlen (also ohne Rücksicht auf Vorzeichenbits). Wir zählen die Zahlen 128 und 130 zusammen:

```
  128       10000000
+ 130     + 10000010
--------------------
  258    (1)00000010
```

Das Ergebnis 258 ist richtig - auch in der Binärdarstellung - nur es paßt nicht mehr in 8 Bits. Ein Bit wurde überTRAGEN in ein extra dafür vorgesehenes Plätzchen: In das Carry-Bit. Jedesmal also, wenn so ein Übertrag in einer Rechenoperation des C 64 stattfindet, zeigt die Carry-Flagge eine 1 (Bild 11).

Je nach Art der von uns programmierten Aufgabe können wir nun dieses Carry-Bit weiterverarbeiten. Es gibt Situationen, in denen man es einfach ignorieren darf (dazu kommen

Befehlswort	Adressierung	Byteanzahl	Code Hex	Dez	Dauer in Taktzyklen	Beeinflussung von Flaggen
INX	implizit	1	E8	232	2	N,Z
INY	implizit	1	C8	200	2	N,Z
INC	absolut	3	EE	238	6	N,Z
DEX	implizit	1	CA	202	2	N,Z
DEY	implizit	1	88	136	2	N,Z
DEC	absolut	3	CE	206	6	N,Z
SED	implizit	1	F8	248	2	1 → D
CLD	implizit	1	D8	216	2	0 → D
BNE	relativ	2	D0	208	2 +1 bei Verzweigung +2 bei Überschreiten einer Seitengrenze	—

Tabelle 2. Die neuen Befehle

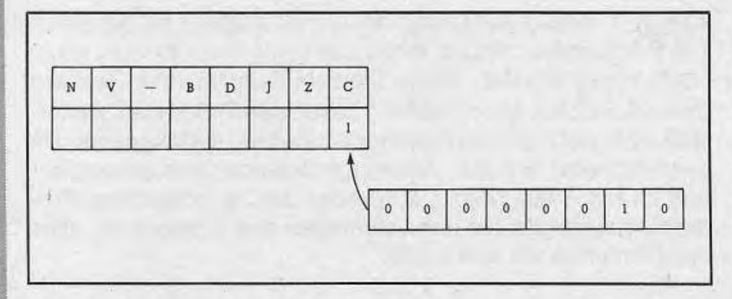


Bild 11. Das Carry-Bit als Bit 8 einer Rechenoperation

wir gleich noch) oder aber solche, wo man es weiter in der Rechnung verwendet. Schließlich kann es auch noch einen Fehler anzeigen: Dann nämlich, wenn das größte zulässige Ergebnis 1111 1111 sein darf. Natürlich kann das Carry-Bit auch gesetzt werden, wenn man in der Zweierkomplementform rechnet. Die Verhältnisse sind dann aber für ein leicht überschaubares Beispiel des Übertrages zu verwickelt, wie Sie gleich sehen werden.

Wenn wir nämlich mit den Zweierkomplement-Zahlen rechnen, dann interessieren uns auch Fälle wie bei der Addition von 64 und 66:

```
     64      01000000
+    66    + 01000010
---------------------
  (-126)     10000010
```

Das ist offensichtlich falsch. Bei der Addition ist durch das Zusammenzählen der Bits 6 plötzlich Bit 7 gesetzt worden. Da wir es aber mit einer Zweierkomplementzahl zu tun haben, bei der dieses Bit 7 eine negative Zahl anzeigt, folgt ein Fehler. Es ist also von Bedeutung, so einen Überlauf (englisch: 'overflow') erkennen zu können um eine entsprechende programmtechnische Reaktion zu starten. Es wird die Überlauf-Flagge V auf 1 gesetzt. Leider ist die Sache aber nicht so einfach, daß sie immer gesetzt würde, wenn von Bit 6 nach Bit 7 ein Übertrag stattfindet. Gesetzt wird diese V-Flagge nur in folgenden zwei Fällen:

1) Es findet ein Übertrag von Bit 6 nach Bit 7 statt, aber kein äußerer Übertrag (wie beim Carry)

2) Es findet kein interner Übertrag von Bit 6 nach Bit 7 statt, aber ein äußerer Übertrag passiert.

Merken kann man sich das am besten so: Immer dann, wenn gewissermaßen das Vorzeichenbit 7 »versehentlich« verändert wurde, wird die V-Flagge auf 1 gesetzt. Das ist ein harter Brocken! Wir sind es ja gewohnt, daß wir uns um diese Dinge beim Computer eigentlich gar nicht mehr kümmern müssen. Außerdem würde das ja erfordern daß man sich bei allen Operationen vorher überlegen muß, welche Fehler also durch »versehentliches« Vorzeichenändern passieren können! Genauso ist es – in der Programmierpraxis wird Ihnen aber das ganze Problem nicht mehr so groß vorkommen. Wir wollen uns dieses Zusammenspiel der Überträge von Bit 6 nach Bit 7 und von Bit 7 nach Bit 8 (also in Carry-Bit) noch anhand einiger Beispiele klarer machen.

Im obigen Beispiel der Addition von 64 und 66 haben wir einen Fall schon behandelt: Es fand ein Übertrag von Bit 6 nach Bit 7 statt, aber kein äußerer Übertrag in Carry-Bit. Deswegen wurde dann auch die V-Flagge gesetzt. Das Problem läßt sich hier ganz einfach lösen zum Beispiel durch Verwendung von 16-Bit-Zahlen:

```
   64      0000000001000000
+  66    + 0000000001000010
---------------------------
  130      0000000010000010
```

Bei 16-Bit-Zahlen ist ja Bit 15 das Vorzeichenbit, welches hier keine Änderung erfährt.

Der andere Fall tritt auf bei der Addition von zwei negativen Zahlen wie -125 und -64:

```
 -125       10000011
 - 64       11000000
--------------------
 (+67)   (1)01000011
```

Auch das ist offensichtlich falsch: Es hat wieder »versehentlich« ein Vorzeichenwechsel stattgefunden. Dies ist also der Fall, wo zwar ein Übertrag ins Carry-Bit stattfand aber kein Übertrag von Bit 6 nach Bit 7. Auch dieses Problem läßt sich durch Verwendung von 16-Bit-Zahlen lösen. Eine kleine Trainingsaufgabe für Sie!

Man kann also sagen: Immer dann, wenn bei 8-Bit-Rechnungen der mittels Zweierkomplementzahlen darstellbare Bereich (127 bis -128) über- oder unterschritten wird, fuhrwerkt man im Vorzeichen-Bit herum und verfälscht das Ergebnis. Dann leuchtet wie eine rote Ampel die Überlauf(V)-Flagge auf und sagt uns, daß wir besser in diesen Fällen mit 16-Bit-Zahlen arbeiten sollten.

Nun noch zum Ignorieren des Carry-Bits, das ich weiter oben erwähnt habe. Bei allen 8-Bit-Rechenoperationen mit Zweierkomplementzahlen kann das Carry-Bit vernachlässigt werden. Zwei Beispiele sollen das wieder illustrieren. Wir addieren +4 und -2:

```
     +4        00000100
+    -2    +   11111110
     +2     (1)00000010
```

Das Carry-Bit wird außer acht gelassen. Anderes Beispiel: Wir addieren zwei negative Zahlen, -4 und -6:

```
     -4        11111010
+    -2    +   11111110
     -6     (1)11111000
```

Auch hier kann man (sogar: muß man) das Carry-Bit vernachlässigen. Beide Ergebnisse sind richtig.

Nun wissen Sie alles über die Art, wie unser Rechner mit ganzen Zahlen arbeitet. Probieren Sie mal ein paar Aufgaben aus zur Übung.

Wir verlassen jetzt den Zahlendschungel und widmen uns der Praxis.

17. Der Computer rechnet: ADC, CLC

ADC ist der erste Arithmetik-Befehl des 6502 (und natürlich auch des 6510), den wir kennenlernen. Er bedeutet »ADd with Carry«, also »addiere mit Carry-Bit«. An einem 8-Bit-Beispiel wollen wir uns das mal ansehen. ZAHL1 und ZAHL2 sollen addiert werden. Beide sollen positive 8-Bit-Zahlen sein, die so klein sind, daß kein Überlauf zu erwarten ist. Die ZAHL1 wird in den Akku gegeben:

```
LDA #ZAHL1
```

Wenn wir nun den Befehl

```
ADC #ZAHL2
```

folgen lassen, sorgt die ALU (arithmetisch-logische Einheit, siehe Folge 1) dafür, daß beide Zahlen addiert werden und das Ergebnis im Akku erscheint. ZAHL1 ist dann vom Ergebnis überschrieben worden. An sich ist damit alles erledigt. Weil wir aber häufig wissen wollen, was denn nun bei der Addition herausgekommen ist, speichern wir den Akku-Inhalt noch irgendwo ab mittels »STA Speicherstelle«. Außerdem war da ja noch die Sache mit dem Carry-Bit. Wir haben oben festgestellt, daß bei einer 8-Bit-Addition kein Carry-Bit berücksich-

tigt werden soll. Nun gibt es aber eine ganze Menge von Vorgängen in einem Assembler-Programm, die das Carry-Bit beeinflussen. Man kann eigentlich vor einer Addition nie ganz sicher sein, ob es denn nun 1 oder 0 ist. Weil jedoch ADC auch das Carry-Bit mitaddiert, sollte man dafür sorgen, daß es vor dem Zusammenzählen wirklich gelöscht ist. Dazu gibt es den Befehl CLC was die Abkürzung für »CLear Carry«, also »lösche Carry-Bit« ist. Sei ZAHL1=12 und ZAHL2=7, dann würde unser vollständiges 8-Bit-Additions-Programmchen also lauten:

```
1200  CLC
1201  LDA   #$0C
1203  ADC   #$07
1205  STA   1500
```

Sehen wir mal davon ab, daß dieses Programm natürlich unsinnig ist (das kann man ja im Kopf schneller rechnen!), dann erkennen wir: CLC ist ein 1-Byte-Befehl mit impliziter Adressierung, welcher sich nur auf die C-Flagge (also das Carry-Bit) auswirkt. ADC ist in der hier verwendeten Form ein 2-Byte-Befehl und liegt in der »unmittelbar« genannten Adressierung vor. Wie wir oben gesehen haben, kann ADC – je nach Art der Rechnung – auf einige Flaggen wirken: Da wären zunächst natürlich die V-Flagge und die C-Flagge. Dann aber kann beim Auftreten eines gesetzten Bit 7 auch die N-Flagge und beim Überschreiten von $FF eventuell auch die Z-Flagge verändert werden.

Viel interessanter wird unser Mini-Programm schon, wenn man anstelle von

```
1201  LDA   #$0C
```

jetzt die absolute Adressierung verwendet, zum Beispiel

```
1201  LDA   1400
```

Weil das ein 3-Byte-Befehl ist, verschiebt sich natürlich der Rest des Programmes um 1 Byte. So kann man immerhin schon zu unterschiedlichen Inhalten von 1400 den gleichen Betrag addieren.

Am interessantesten allerdings ist die Tatsache, daß auch ADC absolut adressierbar ist. Wir können so zum Beispiel den Inhalt der Speicherzelle 1300 zum Inhalt der Zeile 1400 hinzuzählen und dann das Ergebnis in 1500 ablegen:

```
1200  CLC
1201  LDA   1400
1204  ADC   1300
1207  STA   1500
```

Hier ist der ADC-Befehl dann 3 Byte lang geworden.

Vergessen Sie bitte nicht – das gilt vor allem für die nachfolgenden Rechenoperationen – dann, wenn die Wahrscheinlichkeit besteht, daß der Dezimal-Modus eingeschaltet ist (also die D-Flagge auf 1 gesetzt ist), noch den Befehl CLD vor solche Programme zu stellen.

Solche 8-Bit-Rechnungen kommen recht häufig vor: Wenn man in Schleifen nicht mit mehrfach wiederholten INX (beziehungsweise INY oder INC, DEX, DEY oder DEC) arbeiten will, addiert man eben immer die Sprungweite mittels ADC hinzu. Der Akku kann nicht als Zähler dienen, denn es gibt für ihn keinen Befehl, der dem INX und so weiter vergleichbar wäre, weswegen man ihn – sollte es nötig sein – mittels ADC hochzählt.

Häufiger und in der Praxis bedeutender sind 16-Bit-Rechnungen. Wie Sie sicher noch aus den vorangegangenen Folgen wissen, teilt man so eine 16-Bit-Zahl auf in zwei Byte (das LSB und das MSB). Nehmem wir für unser nachfolgendes Beispiel wieder an, daß die Zahlen so gebaut sind, daß kein Überlauf zu befürchten ist. ZAHL1 hätten wir vorher in die Speicherstellen 1300 (LSB) und 1301 (MSB) gepackt, ZAHL2 liegt in den Zellen 1400 (LSB) und 1401 (MSB). Zunächst wieder die Vorbereitungsmaßnahmen:

```
1200  CLD
1201  CLC
```

Dabei ist CLD nicht immer nötig, wie schon gesagt. Nun addieren wir zuerst die LSBs:

```
1202  LDA   1300
1205  ADC   1400
1208  STA   1500
```

Ein Überlauf kann hier nicht stattgefunden haben, denn das Vorzeichenbit ist ja im MSB als Bit 15 enthalten, wohl aber kann ein Übertrag stattgefunden haben: Das Ergebnis könnte größer als 255 ($FF) gewesen sein. War das der Fall, dann ist jetzt eine 1 im Carry-Bit. Wir addieren nun die MSBs:

```
120B  LDA   1301
120E  ADC   1401
1211  STA   1501
```

Egal, was im Carry-Bit stand: Es wurde jetzt hinzuaddiert. Das Ergebnis unserer Rechnung steht nun in 1500 (LSB) und 1501 (MSB). Sehen wir uns das ganze nochmal am Zahlenbeispiel an. Wir addieren die Zahlen 2176 (binär: 0000 1000 1000 0000) und 1009 (binär: 0000 0011 1111 0001). Die Speicherinhalte sind dann:

```
1300  10000000  LSB Zahl1
1301  00001000  MSB
1400  11110001  LSB Zahl2
1401  00000011  MSB
```

Jetzt addieren wir die LSBs:

```
1300        10000000
1400        11110001
Carry              0
--------------------
1500        01110001
Carry:             1
```

Nun folgt der zweite Teil der Addition mit den MSBs:

```
1301        00001000
1401        00000011
Carry:             1
--------------------
1501        00001100
```

Damit steht nun das Ergebnis 3185 (binär 0000 1100 0111 0001) säuberlich aufgeteilt in LSB (Speicher 1500) und MSB (Speicher 1501) fest. Das Carry-Bit steht auch nach vollendeter Rechnung noch auf 1, so daß es vor erneuter Addition wieder mit CLC zu löschen ist.

Damit wäre alles über die Addition berichtet. Wie immer in Programmiererkreisen die Empfehlung: üben, üben,....

Wir wenden uns jetzt der gegenläufigen Operation zu: der Subtraktion.

18. Noch mehr Rechnen: SBC, SEC

Daß das Abziehen von Zahlen im Computer durch das Hinzuzählen des Zweierkomplementes geschieht, haben wir mit viel Gehirnschmalzverbrauch schon in vorangegangenen Abschnitten erfahren. Nun sollen Sie die dazu nötigen Befehlsworte des Assemblers kennenlernen. Zunächst einmal ist da SBC. Das heißt »SuBtract with Carry« oder auf deutsch etwa »ziehe unter Berücksichtigung des Carry-Bits ab«. Ebenso wie bei der Addition mit ADC, wirkt das Argument des SBC-Befehls auf den Akku-Inhalt ein – wobei das Ergebnis im Akku landet, diesen also überschreibt. Komplizierter ist hier die Verwendung des Carry-Bits, worauf wir aber nicht detailliert eingehen wollen. (Wen es interessiert: Nachlesen in L.A. Leventhal, »6502 Programmieren in Assembler«, 3. Auflage, München 1983, Seite 3-100). Für uns soll einfach die nicht ganz korrekte Analogie zum »Borgen« bei der Subtraktion ausreichen. Für den Fall, daß ein solches Borgen eintreten muß, sollte auch das dazu nötige Carry-Bit vorhanden

sein (also auf 1 gesetzt sein). Wie Sie sicherlich schon erraten haben, heißt SEC »SEt Carry«, also »setze das Carry-Bit« (auf 1).

Merke: Vor einer Addition immer Löschen des Carry-Bits mit CLC,
vor einer Subtraktion immer Setzen des Carry-Bits mit SEC!

Zwei Beispiele für die Subtraktion sollen das bisher Gesagte erläutern: Zunächst eine 8-Bit-Subtraktion von ZAHL1 (in Speicherzellle 1300) minus ZAHL2 (in Zelle 1400). Das Ergebnis wird nach 1500 geschrieben:

```
1200  CLD
1201  SEC
1202  LDA   1300
1205  SBC   1400
1208  STA   1500
```

SBC kann – wie hier – absolut adressiert werden, aber auch unmittelbar (also zum Beispiel SBC #$40). Der Befehl ist dann im ersten Fall ein 3-, im anderen Fall ein 2-Byte-Befehl. SEC ist ebenso wie vorher schon CLC ein implizit adressierbarer 1-Byte-Befehl.

Das zweite Beispiel ist eine 16-Bit Subtraktion. In den Speichern steht vor dem Aufruf dieser kleinen Routine:

```
1300  ZAHL1  LSB
1301  ZAHL1  MSB
1400  ZAHL2  LSB
1401  ZAHL2  MSB
```

Das Ergebnis soll nach 1500 (LSB) und 1501 (MSB) gebracht werden:

```
1200  CLD
1201  SEC
1202  LDA   1300
1205  SBC   1400
1208  STA   1500
```

Jetzt sind die beiden LSBs voneinander abgezogen und die Differenz abgespeichert als LSB des Ergebnisses.

```
120B  LDA   1301
120E  SBC   1401
1211  STA   1501
```

Damit ist die Aufgabe beendet. Auch die MSBs sind subtrahiert und das MSB des Ergebnisses steht in 1501.

SBC beeinflußt die gleichen Flaggen wie der Befehl ADC.

19. Ein Programmprojekt

Damit die so kennengelernten Arithmetik-Befehle nicht so trocken auf weiter Flur stehen, wollen wir nun ein Programm entwickeln, aus dem zweierlei zu lernen ist:

1) Die Anwendung bisher gelernter Befehle und
2) ein häufig angewendetes Verfahren, Assemblerprogramme in Basic-Programme einzubinden.

Besonders dieser zweite Aspekt scheint noch vielen Lesern unklar zu sein (das zeigen mir Zuschriften). Es gibt eine ganze Reihe von Möglichkeiten, zum Einbau von Assembler-Routinen in Basic-Programme; die werden wir alle nach und nach kennenlernen. Von Ihnen wurde der SYS-Befehl sicherlich schon häufig angewendet (zum Beispiel für SYS 58640 und vorherigem POKE214,Zeile und POKE211, Spalte zum Setzen des Cursors an die Stelle Zeile, Spalte). Damit haben Sie ein Maschinenprogramm aufgerufen, das im System unseres Computers schon enthalten ist. 58640 ist die Startadresse des Programmes und man kann diesen SYS-Befehl eigentlich wie eine Art »GOTO Maschinenprogramm-Startadresse« ansehen. Nichts hindert uns also, auf diese Weise eigene Assembler-Programme anzuspringen! Das Problem liegt nun nur noch darin, wie man Parameter, die unsere Maschinenroutine benötigt, übergeben kann. Eine offensichtliche – aber leider auch relativ langsame – Methode ist das POKEn der Werte im LSB/MSB-Format in die Speicherzellen, aus denen sie sich unser ML-Programm dann abholt. Wir wollen dieses Verfahren nun an einem Programmbeispiel verwenden.

Eine arithmetische Reihe werden viele von Ihnen schon kennen. Wenn man A als erstes Glied, D als Differenz und N als die Anzahl der Glieder bezeichnet, dann ist die Summe einer solchen Reihe:

$$S=A+(A+D)+(A+2*D)+.......+(A+(N-1)*D)$$

Ein Beispiel ist die Summe der ersten zehn ganzen Zahlen:

$$S=1+2+3+4+5+6+7+8+9+10$$

Hier ist A=1, D=1 und N=10. Daß die Summe S im Beispiel 55 ist, kann man schnell berechnen, was aber, wenn wir wesentlich mehr als nur zehn Glieder haben? Es gibt natürlich auch Formeln zur Berechnung von S. Aber eigentlich ist es ganz reizvoll, ohne solche Formeln den Computer die Summe bilden zu lassen. Wir bauen also ein Programm zur Berechnung der Summe der ersten N ganze Zahlen, wobei N frei gewählt werden kann. Das Ergebnis soll eine 16-Bit-Zahl sein, also nicht größer als 32767. Das beschränkt uns bei N auf Werte von 1 bis 255 (Warum, können Sie ja mal mit dem fertigen Programm ausprobieren). N benötigt also nur 1 Byte Speicherplatz und soll in $1300 abrufbar sein. A soll 1 sein ebenso wie D. Für eventuelle Programmänderungen ist es aber sinnvoll, A und D als 16-Bit-Zahlen aufzubewahren und zwar in $1310/1311 (A in LSB/MSB-Format) und in $1320/1321 (D im gleichen Format). Das Ergebnis soll in $1400/1401 zu finden sein. Das Maschinenprogramm legen wir nach $1200.

Zuerst kümmern wir uns um das Basic-Aufrufprogramm:

Zu diesem Programm gibt es nur noch zu bemerken, daß die Zahlen bei POKE, PEEK oder SYS die Dezimalwerte unserer oben gewählten Adressen sind.

Nun endlich zum Assemblerprogramm. Sehen Sie sich dazu bitte das Flußdiagramm im Bild 12 an.

Wir bereiten den Ablauf vor, indem wir aus $1300 die Anzahl der Glieder ins X-Register laden und zur Vorbereitung der Addition das Carry-Bit löschen. Schalten Sie also bitte den SMON ein und tippen Sie A1200 <RETURN>. Es erscheint die Startadresse 1200. Jetzt können Sie Zeile für Zeile das Assembler-Programm eingeben (nach jeder Zeile ein RETURN, das die nächste Zeilennummer erzeugt):

```
1200  LDX   1300
1203  CLC
```

Die nächsten sechs Zeilen summieren jeweils das neueste Glied zur bis dahin erzeugten Summe. Jetzt zu Beginn ist

```
10  REM **AUFRUF SUMME ARITHMETISCHE REIHE**
20  POKE5120,0:POKE5121,0:REM ERGEBNISSPEICHER AUF
    NULL
30  PRINTCHR$(147)CHR$(17)CHR$(17)
40  INPUT"ANZAHL DER GLIEDER N=";N
50  IFN<1 OR N>255 THEN PRINT CHR$(17)"1 <=N
    <=255":GOTO40
60  POKE4864,N:REM EINSPEICHERN VON N
70  POKE4880,1:POKE4881,0:POKE4896,1:
    POKE4897,0:REM EINSPEICHERN VON A UND D
80  SYS4608:REM AUFRUF UNSERES MASCHINEN-
    PROGRAMMES
90  M=PEEK(5121):L=PEEK(5120):REM AUSLESEN DES
    ERGEBNISSES
100 E=256*M+L:PRINTCHR$(17)CHR$(17)
110 PRINT"DIE SUMME DER ERSTEN "N" GANZEN ZAHLEN
    IST:":PRINTE
120 END
```

$1400/1401 noch leer und in $1310/1311 steht noch das Anfangsglied A=1. Später mit Durchlaufen der Schleife, steht in $1400/1401 immer die bis dahin gebildete Summe und in $1310/1311 das letzte Glied der Reihe. Es handelt sich um die oben kennengelernte 16-Bit-Addition:

```
1204  LDA   1400
1207  ADC   1310
120   ASTA  1400
```

Das neue LSB ist berechnet und in $1400 geschrieben.

```
1200  LDA   1401
1210  ADC   1311
1213  STA   1401
```

Das war nun noch das neue MSB. Als nächstes berechnen wir das momentan letzte Glied der Reihe durch Addieren von D zum alten letzten Glied. Das entspricht dem Basic-Befehl A=A+D in einer Schleife. Dies ist eine neue 16-Bit-Addition, weshalb wir wieder CLC vorgeben müssen:

```
1216  CLC
1217  LDA   1310
121   AADC  1320
121   DSTA  1310
```

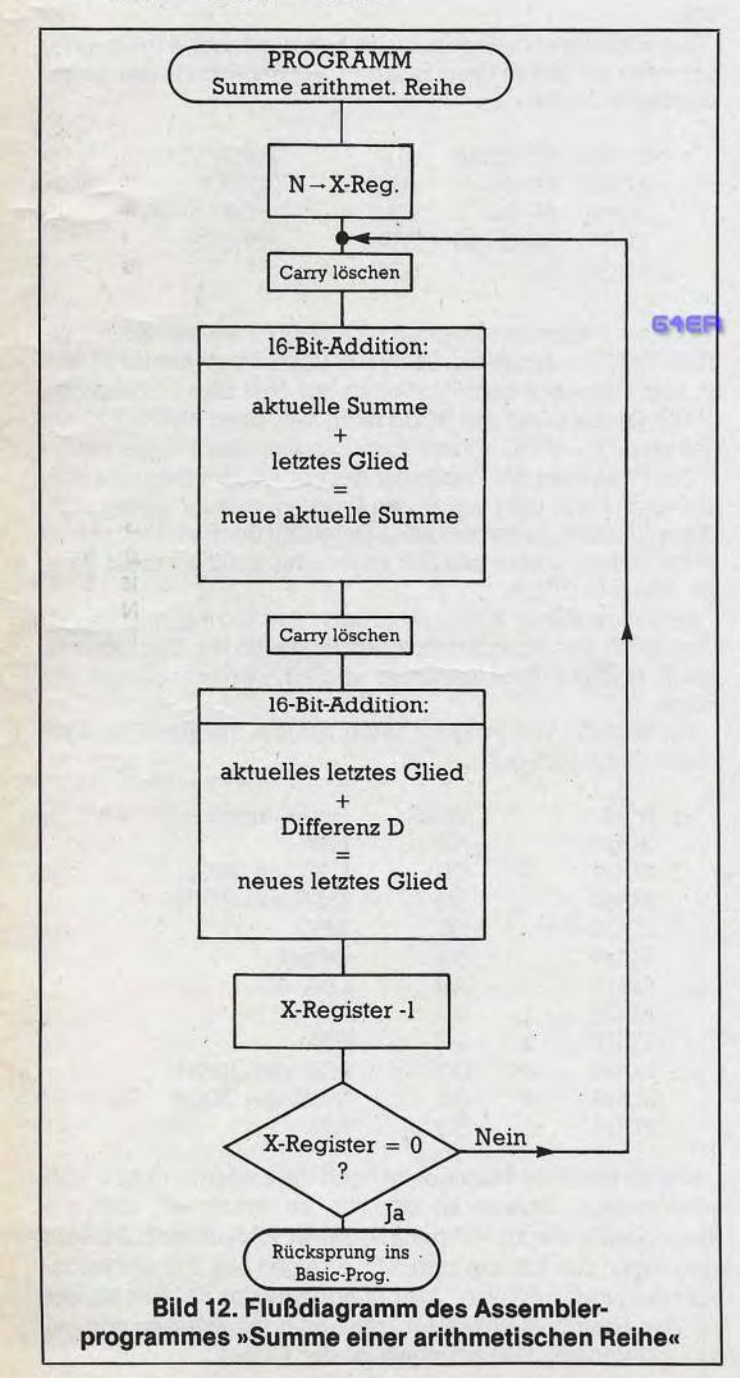


Bild 12. Flußdiagramm des Assemblerprogrammes »Summe einer arithmetischen Reihe«

Das war wieder das LSB. Nun zum MSB:

```
1220  LDA   1311
1223  ADC   1321
1226  STA   1311
```

Wir zählen nun das X-Register um 1 herunter und prüfen, ob es schon Null geworden ist, ob also schon alle N-Glieder summiert worden sind:

```
1229  DEX
122   ABNE  1203
```

Wenn noch nicht alle Glieder berechnet und summiert sind, kehren wir an den Schleifenanfang zurück. Ansonsten springen wir zurück ins Basic-Aufrufprogramm:

```
122   CRTS
```

Wenn Sie beide Programme eingetippt haben, dann speichern Sie sie vorsichtshalber ab (das Assemblerprogramm mit dem S-Befehl des SMON). Beim neuen Einladen brauchen Sie den SMON nicht mehr. Nach dem Laden unseres Maschinenprogrammes (mit ,8,1 bei Diskette oder ,1,1 bei Kassette) geben Sie NEW <RETURN> ein, damit die Zeiger vor dem Einladen des Basic-Programmes wieder auf Normalwerte gesetzt werden. Zwischen dem dann eingeladenen Basic-Programm und unserer Assembler-Routine ist genug Platz. Sollten Sie aber irgendwann mal das Basic-Programm vergrößern, schützen Sie bitte unseren Bereich ab $1200.

Unser Assembler-Beispiel ist so aufgebaut, daß auch A und D variabel gehalten sind. Sie müßten dann nur noch Eingabemöglichkeiten im Basic-Programm schaffen und anstelle der Werte 1 oder 0 in Zeile 70 die LSBs und MSBs der von Ihnen eingegebenen Größen A und D einPOKEn. Auf diese Weise sind dann beliebige ganzzahlige, arithmetische Reihen berechenbar, wie zum Beispiel S=7+10+13+16+... und so weiter. Das überlasse ich Ihrer Basic-Programmierfertigkeit. Nur eines noch: Sie müssen darauf achten, daß die Summe S nicht größer als 32767 wird. Ihrer Phantasie sind – wie immer in diesem Metier – keine Grenzen gesetzt. Sie könnten sich ja mal überlegen, wie man größere Summen zulassen kann (wer sagt denn, daß wir Zahlen immer nur in 2 Byte darstellen dürfen?). Oder Sie könnten sich überlegen, welches eindeutige Merkmal auftritt, sobald der Maximalwert überschritten wird (ein Tip: Lesen Sie doch mal den Abschnitt über die V-Flagge nach).

20. Die Branch-Befehle

Der 6502 (und auch der damit identische 6510) kennt acht bedingte Verzweigungen, von denen wir bisher BNE schon verwendet haben. Alle diese Branch-Befehle (von branch = verzweigen) prüfen Flaggen des Statusregisters.

BNE und BEQ beziehen sich auf die Z-Flagge, die anzeigt, ob im Verlauf der letzten Operation eine Null aufgetreten ist. Ist das der Fall, steht in der Z-Flagge eine 1. BNE verzweigt zur angegebenen Adresse, wenn in der Z-Flagge eine 0 enthalten ist. BEQ (»Branch if EQual zero« = »verzweige, wenn gleich Null«) tut das dann, wenn die Z-Flagge auf 1 gesetzt ist. Da muß man etwas aufpassen, daß man sich nicht vertut!

BCC und BCS haben ihre Aufmerksamkeit auf die C-Flagge, also das Carry-Bit gerichtet. BCC kommt vom englischen »Branch if Carry Clear«, was heißt: »verzweige, wenn das Carry-Bit gelöscht ist«. Ein gesetztes Carry-Bit (also Inhalt=1) veranlaßt BCS (»Branch if Carry Set« = verzweige, wenn das Carry-Bit gesetzt ist) zum Sprung an die angegebene Adresse.

Diese vier bedingten Verzweigungen sind an sich die bedeutsamsten und am häufigsten verwendeten Branch-Befehle. Man kann wohl getrost sagen, daß über 90% der von Programmierern verwendeten bedingten Sprünge, damit absolviert werden. R. Mansfield warnt sogar ausdrücklich in

seinem Buch »Machine language for beginners«, einem in den USA sehr verbreitetem Werk, vor der Verwendung der Befehle BPL und BMI!

Dafür liegt absolut kein einsehbarer Grund vor. Viele programmtechnischen Aufgabenstellungen lassen sich elegant und leicht mit BPL, BMI, BVS und BVC lösen. Man muß nur wissen, wie sie funktionieren und - da liegt vermutlich der Hund begraben - man muß auch die Art kennen, wie Zahlen vom Computer behandelt werden. Genau das aber wissen wir und deswegen sollten wir diese Kenntnis für uns auch nutzen. Also ohne Scheu heran an die verfehmten Befehle!

BMI und BPL (»Branch on MInus« = »verzweige, wenn negativ« und Branch on PLus« = »verzweige, wenn positiv«) hängen mit der Negativ-Flagge N zusammen. Das Rätsel dieser Flagge konnte in den vorangegangenen Folgen gelöst werden: Immer dann, wenn bei einer Operation eine Zahl auftrat, deren Bit 7 eine 1 war, wurde die N-Flagge auf 1 gesetzt. Wir wissen jetzt, daß dieses Bit bei 8-Bit-Zahlen das Vorzeichenbit ist. Bit 7 sagte uns bei einer 1, daß eine negative Zahl im Zweierkomplement-Format vorliegt oder aber überhaupt ein Speicherzelleninhalt vorhanden ist, der größer als 0111 1111 = 127 ist. BMI führt zum Sprung in diesem Fall, weil die N-Flagge auf 1 steht. Andernfalls führt BPL zur Verzweigung.

Ebenso einfach sind BVS und BVC zu erklären: Sie beziehen sich auf die V-Flagge, unsere rote Ampel, die Überlauf bei Rechenoperationen anzeigt. Kann es was bequemeres geben zur Behandlung solcher Fehlrechnungen als ein »Branch on oVerflow Set« = »verzweige, falls die Überlauf-Flagge gesetzt (=1) ist« mit BVS? Oder anders herum bei BVC »Branch on oVerflow Clear« = »verzweige bei freier Überlauf-Flagge«. Wenn man - wie Sie jetzt - weiß, unter welchen Umständen diese V-Flagge auf 1 gesetzt wird, sollte man ohne Skrupel BVS und BVC ausgiebig benutzen. Man könnte damit zum Beispiel programmieren, daß die Rechengenauigkeit automatisch von 16-Bit auf 24- oder 32- (oder wie es gerade beliebt) Bit gesteigert wird, ohne daß man sich bei jeder Programmaufgabe Gedanken über das größtmögliche Ergebnis machen muß. Dazu aber ein andermal mehr.

Alle hier vorgestellten Branch-Befehle sind ebenso wie BNE 2-Byte-Befehle, was an der speziellen Art der Adressierung liegt: Der relativen Adressierung. Tabelle 3 zeigt eine Übersicht der neuen Befehle aus den letzten fünf Kapiteln.

Befehlswort	Adressierung	Byteanzahl	Code hex	Code dez	Dauer in Taktzyklen	Beeinflussung von Flaggen
ADC	unmittelbar	2	69	105	2	N,V,Z,C
	absolut	3	6D	109	4	
CLC	implizit	1	18	24	2	0 → C
SBC	unmittelbar	2	E9	233	2	N,V,Z,C
	absolut	3	ED	237	4	
SEC	implizit	1	38	56	2	1 → C
BEQ	relativ	2	F0		2	keine Änderung
BCC	relativ	2	90		2	keine Änderung
BCS	relativ	2	B0		2	keine Änderung
BMI	relativ	2	30		2	keine Änderung
BPL	relativ	2	10		2	keine Änderung
BVC	relativ	2	50		2	keine Änderung
BVS	relativ	2	70		2	keine Änderung
					+1 bei Verzweigung +2 bei Überschreiten einer Seitengrenze	

Tabelle 3: Die 11 neuen Befehle

21. Die relative Adressierung

Als wir den BNE-Befehl das erstemal verwendet haben, stellten wir fest, daß zum Beispiel BNE 1200 nicht — wie eigentlich zu erwarten war — ein 3-Byte-Befehl, sondern ein 2-Byte-Befehl ist. Damals mußten wir uns mit der Bemerkung zufrieden geben, es läge an der besonderen Art der Adressierung, nämlich der relativen Adressierung. Relativ bedeutet ja »bezogen auf etwas«. Wenn wir also beispielsweise BNE 1200 schreiben, liegt es nur an der Benutzerfreundlichkeit des SMON und vieler anderer Assembler, daß dieser die so geschriebene absolute Adresse 1200 in die richtige Form, nämlich die relative umrechnet. In Wahrheit verlangt der 6502 (und natürlich ebenso der 6510) eine Angabe darüber, wieviele Bytes nach vorne oder hinten im Programm er zur weiteren Programmverarbeitung springen (verzweigen) soll. Es gilt nun also, zwei Fragen zu klären:

1. Relativ wozu wird gesprungen und
2. Wie berechnet sich die Angabe, um wieviele Bytes nach vorne oder hinten im Programm der Sprung vollzogen werden soll.

Zur Klärung verwenden wir ein hypothetisches Programmsegment mit einem Sprungbefehl und sehen uns das Disassemblerlisting an:

```
2000  AD 0030  LDA   3000
2003  F0 05    BEQ   200A
2005  A9 00    LDA   #00
2007  8D 0030  STA   3000
200A  60       RTS
```

Dieses Programm-Teilchen lädt den Inhalt der Speicherstelle 3000 in den Akku, überprüft dann, ob dieser Inhalt Null ist und verzweigt beim Vorliegen der Null zum Rücksprung (RTS). Ist der Inhalt von 3000 nicht Null, dann wird 3000 auf Null gesetzt. 3000 könnte zum Beispiel eine Flagge sein.

Der Pfad, dem der Computer bei der Abarbeitung des Programmes folgt, wird durch den Programmzähler vorbereitet. Dieser ist dann, wenn der BEQ-Befehl an der Reihe ist, schon einen Schritt weiter, nämlich im Programmzähler steht dann die Adresse 2005.

Relativ zu dieser Adresse hat dann der Sprung zu erfolgen. Zum Inhalt des Programmzählers muß also die Sprungweite (auch häufig Offset genannt) addiert werden. Soweit zur Frage 1.

Zur Klärung von Frage 2 listen wir uns mal Byte für Byte unser Programm auf:

Byte		Inhalt	Bedeutung
2000		AD	LDA
2001		00	LSB von 3000
2002		30	MSB von 3000
2003		F0	BEQ
2004		05	Offset
2005		A9	LDA #
2006	1	00	
2007	2	8D	STA
2008	3	00	LSB von 3000
2009	4	30	MSB von 3000
200A	5	60	RTS

Neben der Byte-Nummer ist noch die Entfernung zu 2005 geschrieben. Daraus ist deutlich zu erkennen, daß die Sprungweite, die zum Programmzähler addiert wird, 05 sein muß, wenn der Sprung zum RTS erfolgen soll. Für Vorwärts-Verzweigungen gilt also: Von der Adresse des Befehls an, der auf den Branch-Befehl folgt, zählt man die Byte-Anzahl bis zum Sprungziel. Das Ergebnis ist der Offset.

Nun gibt es genauso häufig Rückwärts-Sprünge. In den bisher gezeigten Programmen sind sie mehrmals aufgetreten. Wie berechnet man den Offset in diesen Fällen? Sehen wir uns wieder das Disassembler-Listing eines solchen Programmsegmentes an:

```
1000  A2 00  LDX #00
1002  E8     INX
1003  D0FD   BNE 1002
1005  00     BRK
...
```

Dieses Programmchen tut nichts anderes, als das vorher auf Null gesetzte X-Register hochzuzählen, bis es über 255 läuft (dann tritt ja wieder 0 auf!). Solange der Inhalt des X-Registers ungleich Null ist, erfolgt ein Sprung zurück bis zur INX-Anweisung in Zeile 1002. Erst wenn die Null durch den Überlauf aufgetreten ist, endet das Programm mit einem BRK in Zeile 1005.

Wir wissen schon, daß der Programmzähler beim Verarbeiten des BNE-Befehls auf 1005 steht. Sehen wir uns auch dieses Programm Byte für Byte an:

Byte		Inhalt	Bedeutung
1000		A2	LDX #
1001		00	
1002	3	E8	INX
1003	2	D0	BNE
1004	1	FD	Offset
1005		00	BRK

Wieder ist neben der Bytenummer die Entfernung vom aktuellen Programmzählerstand angegeben. Wir müssen also vom Inhalt des Programmzählers 3 abziehen, um zum INX-Befehl in Byte 1002 zu gelangen. Das kennen wir aber schon aus den vergangenen Ausgaben: Wenn der Computer eine Zahl abzieht, dann addiert er das Zweierkomplement dieser Zahl. Hier soll nun 3 subtrahiert werden. Wir berechnen das Zweierkomplement:

3 = 0000 0011 (binär)

Das Einerkomplement davon ist:

1111 1100

Dann wird eine 1 addiert

1111 1101

Dies ist das Zweierkomplement. In hexadezimal ausgedrückt heißt diese Zahl $FD und ist unser Offset. Für Rückwärts-Verzweigungen gilt also: Von der auf die Branch-Anweisung folgenden Speicherstelle an zählt man die Bytes zurück bis zum Sprungziel. Das Zweierkomplement der sich dadurch ergebenden Byte-Anzahl ist der Offset.

Das sieht reichlich kompliziert aus, aber zum einen haben Sie ja einen ganz freundlichen Assembler und nur in seltenen Notfällen müssen Sie den Offset berechnen. Zum anderen gibt es noch eine Faustregel, mit der man sich das ganze vereinfachen kann. Die soll durch folgendes Schema erläutert werden:

Byte	Inhalt	Offset
...		
1995		F9
1996		FA
1997		FB
1998		FC
1999		FD
2000	BNE	FE
2001	Offset	FF
2002	Programm-zählerstand	
2003		01
2004		02
2005		03
..		

Bei Vorwärtssprüngen ist ohnehin alles klar: Bei einem Sprung nach Adresse 2005 müßte man in vorliegendem Fall einen Offset von 03 eingeben. Bei Rückwärts-Verzweigungen zählt man einfach von $FF an rückwärts bis zur Zieladresse. Eine Verzweigung nach 1996 würde im vorliegenden Fall also einen Offset von $FA erfordern.

Eine Einschränkung der relativen Adressierung können Sie nun auch sofort verstehen, wenn Sie an Zweierkomplementzahlen denken: Der Offset belegt ein Byte. Die größte positive Zahl in einem Byte ist

0111 1111 = +127 = $7F

und die kleinste negative Zahl ist

1000 0000 = —128 =($80)

Es sind keine größeren Vorwärts-Verzweigungen als um 127 Byte möglich, weil in diesem Fall ein Offset größer als $7F, also mit einem Bit 7 gleich 1 nötig wäre, was aber wieder als negative Zweierkomplementzahl verstanden und einen Rückwärtssprung verursachen würde. Ähnliches gilt anders herum: Es ist kein weiterer Rücksprung als um 128 Byte möglich, weil das im Offset zum gelöschten Bit 7 führen würde, also zu einem Offset kleiner als $80, was wiederum anstelle des Rücksprunges eine Vorwärts-Verzweigung herbeiführen würde.

Darauf sollte man achten beim Erstellen eines Assembler-Programmes, daß man nie weitere Rückwärtssprünge als um 128, beziehungsweise Vorwärtssprünge um 127 Byte verlangt. Auch wenn man im Assembler gar nicht auf relative Adressierung Rücksicht nehmen muß, weil der Assembler sich mit den Absolutadressen begnügt, sollte man wissen, daß zum Beispiel folgende Zeile aufgrund dieser Einschränkung nicht möglich ist:

3000 BNE 1000

Die meisten Assembler reagieren auf solch eine Zeile mit einer Fehlermeldung (beim Hypra-Ass mit »Branch too far«) oder so wie der SMON, der klammheimlich die Programmstartadresse statt 1000 einsetzt. Aber es ist doch ärgerlich, wenn man auf dem Papier ein Programm fertig hat und erst beim Eintippen feststellt, daß der Computer das so nicht haben will.

22. Zeropage-Adressierung

Weil wir nun gerade mit der Adressierung so schön in Schwung sind, stelle ich Ihnen noch eine andere vor: Die Adressierung der Zeropage. Was ist die Zeropage? Auf deutsch heißt das Nullseite. Am besten versteht man das, wenn man sich in Erinnerung ruft, wie Adressen in unserem Computer verwaltet werden. Da haben wir doch ein LSB (Least Significant Byte) und ein MSB (Most Significant Byte), zum Beispiel $1F 04 (mit 1F als MSB und 04 als LSB). Nun hat unser C 64 65535 Adressen von $0000 bis $FFFF. Bei den ersten 256 Adressen von $0000 bis $00FF ist das MSB $00. Man nennt so einen 256-Byte-Block eine Seite (engl. page). Weil hier für alle Adressen dieser ersten Seite des MSB Null ist heißt sie Nullseite = Zeropage. Messerscharf werden Sie schließen, daß man die Seite mit den MSBs $01 als erste Seite bezeichnet, die mit den MSBs $02 als zweite Seite und so weiter.

Wenn wir nun zum Beispiel den Akku mit dem Inhalt der Zeropageadresse $00FA laden wollen, dann könnten wir schreiben:

3000 LDA 00FA

Unser Mikroprozessor versteht uns aber auch, wenn wir nur schreiben:

3000 LDA FA

Das ist sie, die Zeropage-Adressierung. Anstelle eines 3-Byte-Befehls ist das jetzt ein 2-Byte-Befehl, was Speicherplatz und vor allem Rechenzeit einspart. Auf diese Weise kann man von den bisher kennengelernten Befehlen folgende adressieren:
LDA, LDX, LDY, STA, STX, STY, INC, DEC, ADC und SBC

Sie können sich merken, daß man (bis auf zwei Ausnahmen, die wir noch kennenlernen werden) alle absolut adressierbaren Befehle auch Zeropage-absolut anwenden kann. Genauere Angaben über die Codes, die Ausführungszeiten und die Beeinflussung der Flaggen (letztere ist identisch mit der absoluten Adressierung) entnehmen Sie bitte der angefügten Tabelle 4.

Zum Thema Geschwindigkeit: Wenn Sie die benötigten Taktzyklen von absolut und von 0-absolut adressierten Befehlen in den Tabellen miteinander vergleichen, werden Sie jeweils einen Unterschied von einem Zyklus feststellen. Das mag Ihnen läppisch vorkommen. Bedenken Sie aber, daß Sie sehr häufig Schleifen programmieren müssen, die mehrere 100 Mal durchlaufen werden, die vielleicht als oft zu verwendende Unterprogramme dienen... Sie werden bald feststellen, daß da schnell beachtliche Zeitunterschiede auftreten können: Für zeitkritische Programme ist die Verwendung der Zeropage-Adressierung dringend geboten.

Dieser Tatsache waren sich leider auch die Schöpfer unseres Betriebssystems und des Basic-Interpreters voll bewußt. Die Zeropage ist nahezu randvoll mit Speicherstellen, in denen sich beide Programmkomplexe tummeln. Fast jede Kernel- und Interpreter-Routine notiert sich irgendwelche Werte auf der Seite Null. Das macht es uns als Assembler-Programmierer nicht gerade leicht, die Zeropageadressierung zu verwenden, wenn wir außerdem den Interpreter oder das Betriebssystem benutzen wollen. Es kann geradezu katastrophale Folgen haben, einige Zeropage-Adressen zu überschreiben. Andere werden ständig neu beschrieben durch das Betriebssystem oder den Interpreter, was unseren eigenen — vielleicht gerade in so einer Speicherzelle gelagerten — Zwischenwerten den Garaus machen würde. Man sollte sich also die ersten 256 Speicherstellen ganz genau ansehen, bevor man sie adressiert oder aber auf das Betriebssystem und den Basic-Interpreter verzichten. Ersteres erleichtern uns Tabellen der Speicherbelegung (zum Beispiel Babel, Krause, Dripke »Das Interface Age Systemhandbuch zum Commodore 64«, Interface Age Verlag, oder »Das Commodore 64 Buch, Band 4, Ein Leitfaden für Systemprogrammierer«, Markt und Technik Verlag) und auch die Serie von Dr. Helmut Hauck »Memory Map mit Wandervorschlägen«, die seit Ausgabe 11/84 im 64'er erscheint.

Ohne Hemmungen dürfen wir nur die Speicherstellen (jedenfalls beim C 64) $02 und $FB bis $FE nutzen. Weil das doch recht mickrig ist, hat jeder Assembler-Programmierer spezielle Tips, welche Zellen er noch mit welchen Vorsichtsmaßnahmen benutzt. Wenn man bestimmte Routinen aus dem Betriebssystem oder dem Interpreter nicht aufruft, bleiben dazugehörige Zeropageadressen unbeeinflußt und sind dann für eigene Zwecke nutzbar. Manchmal ist es notwendig, den alten Zustand einer Adresse nach Beendigung eigener Programme wieder herzustellen, manchmal nicht. Interessant und viel beschrieben in allen möglichen Zeitschriften, Büchern etc. ist die Möglichkeit, die Notizen, die sich das Betriebssystem oder der Interpreter auf der Zeropage macht, zu verändern. Im Prinzip schreibt man damit kleine Teile dieser Großprogramme um oder variiert Tabellenteile davon. Wie schon Dr. Hauck in seiner Serie sagt, geschieht das im Rahmen der »Tricks« mit irgendwelchen POKEs mehr oder weniger blind, weshalb auch bevorzugt Abstürze des Computers dabei festzustellen sind. Warum Abstürze? Na, stellen Sie sich mal ein von Ihnen geschriebenes Programm vor — zum Beispiel das aus Kapitel 19 zur Berechnung der Summe einer arithmetischen Reihe — und POKEn Sie dann anstelle irgendeines Befehlscodes, der dorthin gehört, jetzt eine 0 (also ein BRK) hinein. Die Wirkung dürfte ähnlich sein. Wenn man allerdings die Funktion der betreffenden Speicherstelle genau kennt, lassen sich recht nützliche Änderungen hervorrufen, wie zum Beispiel die Schutz-POKEs für den Basic-Speicher durch Verändern der Adressen $33, $34, $37 und $38.

Wir werden im folgenden immer dann, wenn wir mit Zeropage-Adressierung arbeiten oder Routinen des Betriebssystems oder Interpreters untersuchen, spezielle Stellen der Nullseite kennenlernen.

Vorhin hatte ich noch angedeutet, daß man dann die Zeropage fast vollständig nutzen könne, wenn man auf den Basic-Interpreter und das Betriebssystem verzichtet. Das ist tatsächlich möglich. Nur wird man dann erstaunt feststellen, wieviel Arbeit uns die computerinterne Software abnimmt oder anders herum: Viele bislang selbstverständliche Dinge werden wir dann plötzlich selbst programmieren müssen, und das kann ein hartes Brot sein!

Als Beispiel für ein Programm, das nicht nur die Zeropageadressierung verwendet, sondern sogar selbst komplett in der Zeropage steht, werden wir uns die CHRGET-Routine ansehen. Eine Klasse von Befehlen, die dort angewendet wird, die Vergleichsbefehle, soll zuvor noch gezeigt werden.

Befehlswort	Adressierung	Byteanzahl	Code Hex	Code Dez	Dauer in Taktzyklen	Beeinflussung von Flaggen
LDA	0-Page, abs.	2	A5	165	3	N,Z
LDX	0-Page, abs.	2	A6	166	3	N,Z
LDY	0-Page, abs.	2	A4	164	3	N,Z
STA	0-Page, abs.	2	85	133	3	—
STX	0-Page, abs.	2	86	134	3	—
STY	0-Page, abs.	2	84	132	3	—
INC	0-Page, abs.	2	E6	230	5	N,Z
DEC	0-Page, abs.	2	C6	198	5	N,Z
ADC	0-Page, abs.	2	65	101	3	N,V,Z,C
SBC	0-Page, abs.	2	E5	229	3	N,V,Z,C
CMP	unmittelbar	2	C9	201	2	
	absolut	3	CD	205	4	
	0-Page, abs.	2	C5	197	3	
CPX	unmittelbar	2	E0	224	2	
	absolut	3	EC	236	4	N,Z,C
	0-Page, abs.	2	E4	228	3	
CPY	unmittelbar	2	C0	192	2	
	absolut	3	CC	204	4	
	0-Page, abs.	2	C4	196	3	

Tabelle 4: Kenndaten der neuen Befehle und Adressierungen

23. Die Vergleichsbefehle: CMP, CPX, CPY

Vergleichen heißt in englischer Sprache »to compare«, woraus Sie unschwer erkennen können, woher die Bezeichnung CMP und die CPs in CPX beziehungsweise CPY kommen. Verglichen wird jeweils der Akku-Inhalt (bei CMP), der Inhalt des X- (bei CPX) oder des Y-Registers (bei CPY) mit Daten, die der Compare-Befehl adressiert. Einige Beispiele werden Ihnen das klarer machen:

```
CMP #FF
```

vergleicht den Akku-Inhalt mit der Zahl $FF. Hier liegt die unmittelbare Adressierung vor, die ebenso für CPX und CPY verwendbar ist. Außerdem ist das dann ein 2-Byte-Befehl.

```
CPX 3000
```

vergleicht den Inhalt des X-Registers mit dem Inhalt der Spei-

cherstelle $3000. Die absolute Adressierung ist also auch anwendbar (natürlich auch für CMP und CPY). Der Compare-Befehl besteht so aus 3 Byte.

CPY A8

vergleicht den Inhalt des Y-Registers mit dem Inhalt der Zeropagestelle $A8. Diese soeben frisch gelernte Zeropage-Adressierung ist bei allen drei Vergleichsbefehlen möglich und macht aus ihnen 2-Byte-Befehle.

Für CPX und CPY sind das alle Möglichkeiten der Adressierung. CMP erlaubt weitere, die wir noch kennenlernen werden. Nun interessiert uns natürlich noch, wie das Vergleichsergebnis zu erhalten ist! Bei diesen Befehlen geschieht merkwürdiges: Die Vergleichsdaten werden vom Inhalt des Akkus (beziehungsweise X- oder Y-Registers) abgezogen, aber: Weder wird dieser Inhalt noch werden die adressierten Daten verändert! Der Trick ist, daß drei Flaggen das Ergebnis anzeigen: Die Negativ-Flagge N, die Null-Flage Z und das Carry-Bit C. Diese Anzeige geschieht so: (Bild 13)

FLAGGE	Akku, X, Y } >DATEN	Akku, X, Y } =DATEN	Akku, X, Y } <DATEN
N	0 oder 1	0	1 oder 0
Z	0	1	0
C	1	1	0

Bild 13. Flaggen bei den Vergleichsbefehlen

lsn $	msn	$ bin. binär	0 0000	1 0001	2 0010	3 0011	4 0100	5 0101	6 0110	7 0111
0	0	000	NUL	DLE	SP	0	@	P		p
			NULL	DLE	SP	0	@	P	CHR$(96)	CHR$(112)
1	0	001	SOH	DC1	!	1	A	Q	a	q
			SOH	DC1	!	1	A	Q	CHR$(97)	CHR$(113)
2	0	010	STX	DC2	″	2	B	R	b	r
			STX	DC2	″	2	B	R	CHR$(98)	CHR$(114)
3	0	011	ETX	DC3	#	3	C	S	c	s
			ETX	DC3	#	3	C	S	CHR$(99)	CHR$(115)
4	0	100	EOT	DC4	$	4	D	T	d	t
			EOT	DC4	$	4	D	T	CHR$(100)	CHR$(116)
5	0	101	ENQ	NAK	%	5	E	U	e	u
			ENQ	NAK	%	5	E	U	CHR$(101)	CHR$(117)
6	0	110	ACK	SYN	&	6	F	V	f	v
			ACK	SYN	&	6	F	V	CHR$(102)	CHR$(118)
7	0	111	BEL	ETB	′	7	G	W	g	w
			BEL	ETB	′	7	G	W	CHR$(103)	CHR$(119)
8	1	000	BS	CAN	(	8	H	X	h	x
			BS	CAN	(	8	H	X	CHR$(104)	CHR$(120)
9	1	001	HT	EM	)	9	I	Y	i	y
			HT	EM	)	9	I	Y	CHR$(105)	CHR$(121)
A	1	010	LF	SUB	*	:	J	Z	j	z
			LF	SUB	*	:	J	Z	CHR$(106)	CHR$(122)
B	1	011	VT	ESC	+	;	K	[	k	{
			VT	ESC	+	;	K	[	CHR$(107)	CHR$(123)
C	1	100	FF	FS	,	<	L	\	l	¦
			FF	FS	,	<	L	£	CHR$(108)	CHR$(124)
D	1	101	CR	GS	—	=	M	]	m	}
			CR	GS	—	=	M	]	CHR$(109)	CHR$(125)
E	1	110	SO	RS	·	>	N	↑	n	~
			SO	RS	·	>	N	↑	CHR$(110)	CHR$(126)
F	1	111	SI	US	/	?	O	—	o	DEL
			SI	US	/	?	O	←	CHR$(111)	CHR$(127)

Bild 14: ASCII-Code (jeweils oben) und Commodore-ASCII (msn = most significant nibble; lsn = least significant ni.

1) Der Registerinhalt (Akku, X-, Y-Register) ist größer als die Vergleichsdaten:
Dann ist das Carry-Bit = 1, die N- und die Z-Flagge =0.
2) Der Registerinhalt ist gleich den Vergleichsdaten:
Dann sind Carry- und Z-Flagge = 1, die N-Flagge = 0.
3) Der Registerinhalt ist kleiner als die Vergleichsdaten:
Die N-Flagge ist dann =1, Carry- und Zero-Flagge sind 0.

Damit Sie die Übersicht behalten können, ist in Bild 13 das ganze als Schema gezeigt.

Sie werden sich vermutlich schon denken können, wie der Hase weiterläuft: Mit den Verzweigungsbefehlen prüfen wir die Flaggen und springen die gewünschten weiteren Programm-Routinen an.

Die Kombination der Compare-Befehle mit den Verzweigungsoperationen wird Ihnen im weiteren Verlauf dieses Kurses noch ganz geläufig werden. Ein Beispiel sehen Sie nachher ebenfalls in der CHRGET-Routine. Leider muß ich Sie immer noch etwas vertrösten, denn mit Verstand begreifen läßt sich diese Routine nur dann, wenn man etwas mehr über die Codierung von Zeichen weiß. Deswegen werden wir uns nun noch mit dem ASCII-Code und dem Commodore-ASCII herumschlagen.

24. Zeichencodierung mit dem ASCII- und dem Commodore-ASCII-Code

ASCII ist die Abkürzung von »American Standard Code for Information Interchange« und das heißt auf deutsch »amerikanischer Standard-Code zum Informations-Austausch«. Diese Zeichenverschlüsselungsart ist international als ISO-7-Bit-Code genormt, und es wäre wirklich nett, wenn alle sich daran halten würden. Tatsächlich aber finden wir zum Beispiel bei unserem C 64 eine Abart des Normcodes, den Commodore-ASCII-Code. Über die damit erzwungenen Umrechnungen können alle diejenigen Dramen erzählen, die zum erstenmal einen (Nicht-Commodore-)Drucker an ihr Gerät anschließen oder aber blauäugig in den Online-Betrieb mit anderen Computern eintreten wollten.

Sehen wir uns zunächst einmal den ASCII-Code an. Es handelt sich um einen 7-Bit-Code, das heißt 128 Zeichen können in nur 7 Bit untergebracht werden (0000 0000 bis 01111111). Das achte Bit dient bei manchen Operationen mit Computer-Peripherie als Paritäts-Bit. Bei dieser Gelegenheit soll auch gleich erklärt werden, was Parität in diesem Zusammenhang bedeutet. Werden Daten übertragen, muß immer

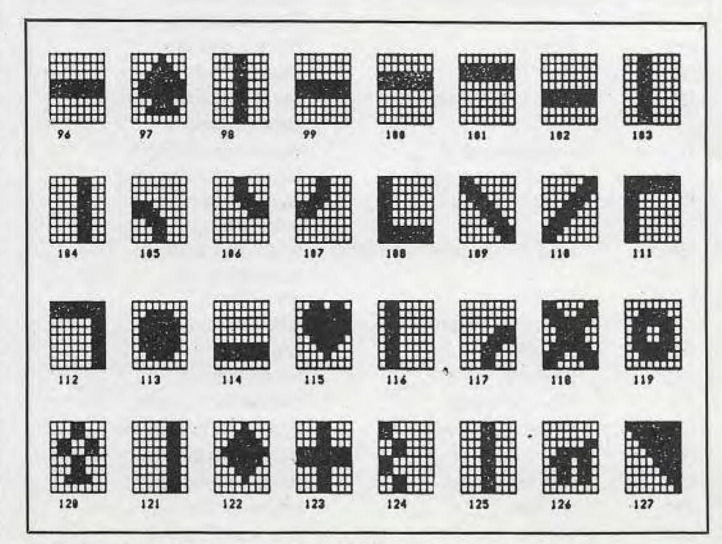


Bild 15. Grafikzeichen zu den entsprechenden CHR$-Codes

mit Übermittlungsfehlern gerechnet werden. Das Paritätsbit dient dazu festzustellen, ob ein Byte korrekt angekommen ist. Bei der sogenannten geraden Parität zählt man die Einser im Byte zusammen und setzt Bit 7 auf 1 wenn sich eine ungerade Zahl ergibt. Mit dem Paritätsbit haben wir dann eine gerade Zahl. Ist die Quersumme des Byte schon gerade, bleibt Bit 7 eine Null. Ebensogut kann man die ungerade Parität verwenden, indem dann Bit 7 so gewählt wird, daß sich immer eine ungerade Zahl ergibt. Welche Art der Parität zur Anwendung kommt, ist Vereinbarungssache. Nehmen wir mal an, es sei gerade Parität gefordert und ein Byte mit der Information 00010110 soll übermittelt werden. Die Quersumme ist 3, also ungerade. Das Paritätsbit muß auf 1 gesetzt werden. Wir senden das Byte 10010110. Der Empfänger überprüft zunächst auf gerade Parität und verwendet dann nur die Bits 0 bis 6. Doppelfehler, die mittels des Paritätsverfahrens nicht festgestellt werden können, sind sehr selten. Leider kann auf diese Weise nur bemerkt werden, daß ein Übertragungsfehler aufgetreten sein muß, aber nicht welcher. Die Information muß dann neu angefordert werden.

Sehen wir uns nun den Commodore-ASCII-Code an. Durch die Einbindung der Grafikzeichen brauchen wir mehr als die 128 Kombinationen. Commodore benutzt deswegen einen 8-Bit-Code. Mit dem Basic-Befehl CHR$(x) können Sie sich alle 256 Möglichkeiten ansehen. Erschwerend kommt aber noch hinzu, daß wir nicht nur einen Zeichensatz, sondern deren vier zur Verfügung haben, die durch den jeweiligen Schreibmodus ansprechbar sind (Klein-/Großschriftmodus, Großschriftmodus, beide Modi mit Reverse-ON oder OFF). Im Zeichen-ROM liegen insgesamt 512 Muster abrufbereit. Zu diesen kommen beim CHR$-Befehl noch eine ganze Reihe von Steuerzeichen hinzu... die Verwirrung ist perfekt! Wir wollen an dieser Stelle keine Entwirrung vornehmen, sondern wir durchschlagen den Gordischen Knoten, indem wir nur die ersten 128 Zeichen mit den ASCII-Zeichen vergleichen. In Bild 14 und 15 finden Sie unsere Gegenüberstellung.

Einige Kombinationen dienen als Steuer-Codes. (Die Bedeutung der dabei verwendeten Abkürzungen sehen Sie in Bild 16.)

Nur ein Teil dieser Codes wird tatsächlich genutzt. Andere haben – je nach Gerät an das sie gesandt werden – unterschiedliche Bedeutungen. Denken Sie dabei nur mal an die verschiedenen Betriebssysteme des Commodore-Druckers 1526, wo man bei dem einen mit CHR$(1), bei dem anderen mit CHR$(14) den Breitschrift-Modus anschaltet. Innerhalb unseres Computers werden offensichtlich bestimmte Codes anders genutzt. Das sind:

Anstelle von	geschieht folgendes:
ENQ	Zeichen weiß
BS	Blockieren der Umschaltung Klein-/Großschrift
HT	Zulassen der obigen Umschaltung
DC1	Cursor abwärts
DC2	Reverse-Modus an
DC3	Cursor in HOME-Position
DC4	INST/DEL
FS	Zeichen rot
GS	Cursor rechts
RS	Zeichen grün
US	Zeichen blau

Der auffälligste Unterschied ist der, daß beim Commodore-ASCII anstelle der Kleinbuchstaben Grafikzeichen liegen. Sollte anstelle des Normalmodus der Klein-/Großschriftmodus eingeschaltet sein, findet man anstelle der Großbuchstaben die kleinen.

Jetzt haben wir alle nötigen Kenntnisse, um die CHRGET-Routine in unserem Computer zu verstehen.

NUL	Null	
SOH	Start of heading	Beginn des Kopfes
STX	Start of text	Textbeginn
ETX	End of text	Textende
EOT	End of transmission	Übertragungsende
ENQ	Inquiry	Anfrage
ACK	Acknowledge	Bestätigung
BEL	Bell	Klingel
BS	Backspace	Zurücksetzen
HT	Horizontal tabul.	Horizontaltabulator
LF	Line feed	Zeilenvorschub
VT	Vertical tabulator	Vertikaltabulator
FF	Form feed	Formatvorschub
CR	Carriage return	Wagenrücklauf/ Zeilenwechsel
SO	Shift out	Rückschaltung
SI	Shift in	Dauerumschaltung
DLE	Data link escape	Datenverbindungs-umschaltung
DC1-4	Device control	Gerätesteuerung
NAK	Negative acknowl.	Negativ-Bestätigung
SYN	Synchronous idle	Synchronisations-Leerlauf
ETB	End of transmission block	Ende des Übertragungsblockes
CAN	Cancel	Annullieren
EM	End of medium	Datenträgerende
SUB	Substitute	Ersetzen
ESC	Escape	Umschaltung
FS	File separator	Dateitrennzeichen
GS	Group separator	Gruppentrennzeichen
RS	Record separator	Satztrennzeichen
US	Unit separator	Einheiten-Trennz.
SP	Space	Leerzeichen
DEL	Delete	Löschzeichen

Bild 16. Die Bedeutung der Abkürzungen im ASCII Code

25. Die CHRGET-Routine

Das Kürzel CHRGET kommt von »Get a character«, was bei uns heißt: »Hole ein Zeichen«. Es handelt sich um eine sehr häufig benutzte Routine unseres Basic-Interpreters, die – wie schon vorhin erwähnt – komplett in der Zeropage steht. Wenn Sie mit dem SMON mal nachsehen wollen, dann geben Sie den Befehl

D 0073 008B

ein. Sie haben dann die komplette Routine vor sich:

```
0073  E6  7A    INC 7A
0075  D0  02    BNE 0079
0077  E6  7B    INC 7B
0079  AD  2502  LDA 0225
007C  C9  3A    CMP #3A
007E  B0  0A    BCS 008A
0080  C9  20    CMP #20
0082  F0  EF    BEQ 0073
0084  38        SEC
0085  E9  30    SBC #30
0087  38        SEC
0088  E9  D0    SBC #D0
008A  60        RTS
```

Eventuell sieht die Zeile 0079 bei Ihnen anders aus. Das liegt dann an den Speicherstellen 7A und 7B, welche einen Zeiger darstellen (LSB=7A und MSB=7B), der bei Ihnen gerade auf einen anderen Platz zeigt als auf $0225.

Diese CHRGET-Routine besteht aus drei Teilen:

Zeilen 0073 bis 0079

Weiterstellen des CHRGET-Zeigers und Einladen des dadurch angezeigten Speicherzelleninhaltes in den Akku.

Zeilen 007C bis 0082

Prüfroutinen

Zeilen 0084 bis 008A

Flaggen-Routinen

Im ersten Teil haben wir schon gleich etwas neues vor uns: ein sich selbst veränderndes Programm. Die Speicherstelle (aus dem Basic-Eingabepuffer), aus der der Akku ein Zeichen holt, wird um 1 weitergezählt mit INC 7A.

Dabei handelt es sich um das LSB der Adresse und die nächste Zeile prüft, ob ein Überlauf (255+1) stattgefunden hat:

BNE 0079.

Diese Technik kennen wir schon aus den letzten Folgen: Bei Überlauf wird die Z-Flagge auf 1 gesetzt und der BNE-Befehl führt keinen Sprung herbei. Den Offset von 02 können wir leicht nachrechnen: Der Programmzähler steht schon auf 0077. Die Zieladresse 0079 ist also noch 2 Byte entfernt. Hat eine Überschreitung des Höchstwertes 255 stattgefunden, dann muß das dazugehörige MSB um 1 erhöht werden. Dies tut die nächste Zeile: INC 7B

In beiden Fällen ist nun der Zeiger 7A/7B um eine Stelle weitergerückt und der Inhalt der dadurch angezeigten Speicherstelle wird in den Akku geladen. Zwei Dinge können wir uns aus diesem kurzen Programmteil merken:

1) Wie man eine 16-Bit-Zahl hoch- (oder auch herunter-) zählt und

2) eine Möglichkeit, Zeiger einzusetzen. Wir werden noch eine Reihe anderer Zeigertypen kennenlernen und sehen, daß es nicht immer so direkt zugeht wie hier.

Im zweiten Teil finden wir die Prüfroutinen. Die Vergleichsbefehle beschränken sich auf den Akkuinhalt, also CMP.

CMP #3A testet, in welcher Beziehung das im Akku befindliche Zeichen zum Wert $3A = dezimal 58 steht. Erinnern wir uns an das Schema in Bild 14:

1) Commodore-ASCII-Code im Akku größer als 58, also Zeichen hinter dem Doppelpunkt (Buchstaben, Grafikzeichen, einige Sonderzeichen). Dann ist die Carry-Flagge = 1, N- und Z-Flagge sind 0.

2) Im Akku steht genau der Code 58, also der Doppelpunkt. Dann sind Carry-Bit und Z-Flagge = 1, nur die N-Flagge = 0.

3) Der Code des Zeichens im Akku ist kleiner als 58 (das wären alle Zahlen, einige Sonderzeichen und Steuerzeichen). In diesem Fall ist die N-Flagge = 1. Die beiden anderen Flaggen zeigen Null.

Der nun folgende Befehl BCS 008A überprüft die Carry-Flagge. Wenn sie gesetzt ist, wenn also der Code im Akku größer oder gleich dem eines Doppelpunktes (58) ist, springt der Programmzähler zum RTS. Der Code (und auch die Flaggen) wird unverändert zum aufrufenden Hauptprogramm weitergegeben. Zur Übung können Sie ja nochmal den Offset nachrechnen. Der Rest des Programms wird nur noch durchlaufen, wenn Codes kleiner als 58 im Akku stehen.

Die nächste Zeile CMP #20 dient zum Vergleich des Space-Codes $20 = dezimal 32 (Leertaste). Die Flaggen treten dann, wie schon oben beim ersten Vergleich gezeigt, je nach Akku-Inhalt auf. Durch die Verzweigung BEQ 0073 erfolgt ein Rücksprung zum Beginn der CHRGET-Routine dann, wenn die Z-Flagge gesetzt ist, also ein Space-Code im Akku liegt. Somit werden die Leerzeichen einfach übersprungen und das nächste Zeichen geholt. Alle anderen Zeichen, die bis hierher durchgehalten haben, werden nun im letzten Teil der CHRGET-Routine einer Prozedur unterworfen, die ich Flaggen-Routine genannt habe.

Durch zwei aufeinanderfolgende Subtraktionen, die insgesamt den Wert im Akku unverändert lassen (es wird 256 abgezogen), wird die Carry-Flagge beeinflußt. Verfolgen wir, was da passiert:

SEC dient als Vorbereitung für die folgende Subtraktion.

SBC #30 zieht vom Akku-Inhalt $30 = dezimal 48 ab. Wir wissen inzwischen, daß das der Addition des Zweierkomplementes entspricht. Dieses ist (rechnen Sie mal nach!) 1101 0000.

Nehmen wir mal an, wir hätten den Code der Zahl 4 (also dezimal 52 oder $34) im Akku stehen. Die Rechnung sieht dann so aus:

```
  52    0011  0100
        1101  0000
   +  ------------
 (1)    0000  0100
```

Das Ergebnis ist also 4, der Übertrag wird vernachlässigt.

Als anderes Beispiel sei nun der Code für das Ausrufungszeichen im Akku (dezimal 33 = $21 = binär 0010 0001). Die Rechnung ist dann:

```
  33    0010  0001
   +    1101  0000
      ------------
        1111  0001
```

Das Ergebnis ist —15.

Alle Codes, die nicht für Zahlen stehen, haben nach dieser Subtraktion ein negatives Ergebnis im Akku hinterlassen und durch das »Borgen« das Carry-Bit gelöscht.

Nun machen wir weiter ab Zeile 0087:

SEC

SBC #D0

Wir ziehen $D0 = dezimal 208 ab. Das Zweierkomplement ist: ...Doch da kommen wir ins Stocken! Denn dieses Zweierkomplement ist nicht mehr mit 8-Bit-Zahlen darzustellen. Schon die Zahl 208 im Binärformat (1101 0000) würde als negative Zahl angesehen werden, weil Bit 7 gleich 1 ist. Wir machen es uns einfach und sagen, daß sich das Zweierkomplement wie bisher bilden läßt, aber dabei das Carry-Bit mit einbezogen wird. Unser Zweierkomplement ist dann also: 0011 0000 und das Carry-Bit ist gelöscht. Nun nehmen wir unser erstes Beispiel. Dort war nach der Subtraktion im Akku eine 4 verblieben:

```
        0000  0100
   +    0011  0000
      ------------
        0011  0100
```

Das ist wieder unser ursprünglicher Wert dezimal 52 = $34 = Code für die Zahl 4. Das Carry-Bit bleibt gelöscht.

Im zweiten Beispiel mit dem Ausrufungszeichen stand noch im Akku eine —15:

```
        1111  0001
   +    0011  0000
      ------------
 (1)    0010  0001
```

Da haben wir wieder den Code für das Ausrufungszeichen ($21 = dezimal 33) im Akku und ein gesetztes Carry-Bit. Was kommt also bei der CHRGET-Routine heraus?

1) Alle Zeichen außer dem Space werden unverändert an das aufrufende Programm über den Akku weitergegeben. Space wird unterdrückt.

2) Bei allen Zeichen außer bei den Zahlen ist das Carry-Bit gesetzt.

3) Manche der aufrufenden Routinen überprüfen außer dem Zustand der Carry-Flagge auch den der Z- oder N-Flagge, die ja beim ersten CMP-Befehl ebenfalls gesetzt werden. So liefert die CHRGET-Routine noch weitere Informationen.

In der einschlägigen Literatur stoßen Sie auch auf eine Routine, die CHRGOT genannt wird. Es handelt sich dabei ebenfalls um die hier beschriebene CHRGET-Routine, nur erfolgt der Einsprung nicht bei $0073, sondern bei $0079. Der Zeiger $007A/7B wird in diesem Fall nicht weitergestellt. Das vorher schon einmal in den Akku geladene Zeichen wird damit noch einmal angesprochen (got ist die Vergangenheitsform von get).

Mit dem CHRGET-Programm haben wir eines der wichtigsten Unterprogramme unserer computerinternen Software kennengelernt. Will man sich Interpreter-Routinen zunutze machen, stolpert man ständig darüber. Außerdem aber liegt die CHRGET-Routine im RAM. Das bedeutet, daß wir sie ohne weiteres für unsere Zwecke verändern können. Ein Beispiel für so eine Änderung hat Christoph Sauer in seiner Serie über den »gläsernen VC 20« in der Ausgabe 9 (Seite 158) gezeigt. Dort wird die CHRGET-Routine nach dem LDA angezapft und auf das Pi-Zeichen geprüft, das neuen Befehlen vorangestellt wurde. Sehen Sie sich das Programm dort (auf Seite 160f.) mal genau an, viel kann man durch Nachvollziehen fremder Programme für die eigene Programmiertechnik lernen.

26. Die indizierte Adressierung

Indizieren heißt, etwas mit einem Index, also einem Zeichen oder einer Nummer, zu versehen. Beispielsweise bezeichnet man in der Mathematik die beiden Lösungen einer quadratischen Gleichung häufig als X1 und X2. Dabei ist dann die Ziffer (1 oder 2) der Index und X ist eine indizierte Größe. Man geht also aus von einer festgelegten Grundmenge (Lösungsmenge X) und trifft durch den Index eine weitere Unterscheidung.

So ähnlich können wir uns auch die Funktion der indizierten Adressierung bei der Assembler-Programmierung vorstellen. Nehmen wir als Beispiel den Befehl

```
LDA 1500,X
```

Man spricht hier von einer absolut-X-indizierten Adressierung. Das Assemblerwort LDA ist uns bekannt: Lade den Akku. Woher soll der für den Akku bestimmte Inhalt geholt werden? Aus der Speicherzelle, die sich durch 1500 plus Inhalt des X-Registers ergibt. Steht also im X-Register zum Zeitpunkt des Befehlsaufrufes eine 5, dann wird der Akku aus Speicherzelle 1500+5, also 1505, geladen. Das X-Register kann Werte von 0 bis $FF (dez. 255) enthalten. Die Ähnlichkeit sieht also so aus:

Aus einer Gesamtmenge von 256 Adressen, die durch die Anfangsadresse (bei unserem Beispiel 1500) und die möglichen 256 Belegungen des X-Registers festgelegt sind (die Grundmenge), werden je nach X-Registerinhalt einzelne Adressen unterschieden und adressiert. Das X-Register fungiert dabei als ein Index, weswegen man auch oft die Bezeichnung »Index-Register X« in der Literatur findet.

Ebenfalls als Index-Register kann das Y-Register dienen, was zum Beispiel zum Befehl

```
LDX 1500,Y
```

führen kann. Dies ist dann eine absolut-Y-indizierte Adressierung.

Genauso wie man die normale absolute Adresse (also zum Beispiel 1500) als Basis der Indizierung durch das X- oder das Y-Register verwenden kann, ist das auch mit eine Zeropage-Adresse möglich. So gibt es zum Beispiel die Befehle

```
LDY 2B,X
```

oder

```
STX 19,Y
```

Man nennt diese Art der Adressierung dann Zeropage-absolut-X-indiziert beziehungsweise -Y-indiziert.

Weil die Zeropage aber nur 256 Adressen umfaßt, andererseits jedoch die Indexregister auch 256 Werte annehmen können, kann es geschehen (wenn man nicht aufpaßt), daß die Summe aus der Basisadresse (zum Beispiel $2B) und dem Indexregisterinhalt größer als 256 wird. Wenn zum Beispiel in dem Befehl

```
LDA FE,X
```

der X-Registerinhalt 2 beträgt, ergäbe sich $FE+$02=$0100. In diesem Fall wird aber nicht der Inhalt von $0100 in den Akku geladen, sondern der Befehl spricht die Speicherstelle $00 an. Der Grund dafür liegt in der Tatsache, daß unser Prozessor den Befehl als 2-Byte-Befehl interpretiert – das 2. Byte ist die Zeropageadresse, die sich als Summe ergibt – und deswegen nur das LSB der Adresse beachtet. Von $0100 ist das LSB aber $00. Mit anderen Worten: Die Zeropage-absolut-indizierten Befehle lassen einen Zugriff nur auf die Zeropage selbst zu. Dieses Verhalten muß man beim Programmieren beachten.

Wir wollen nochmal zusammenfassen. Vier neue Adressierungsarten haben wir kennengelernt:

Befehl	Indizierte Adressierung			
	absolut		Null-Seite-absolut	
	X	Y	X	Y
LDA	+	+	+	-
LDX	-	+	-	+
LDY	+	-	+	-
STA	+	+	+	-
STX	-	-	-	+
STY	-	-	+	-
RTS	/	/	/	/
INX	/	/	/	/
INY	/	/	/	/
INC	+	-	+	-
DEX	/	/	/	/
DEY	/	/	/	/
DEC	+	-	+	-
SED	/	/	/	/
CLD	/	/	/	/
BNE	/	/	/	/
ADC	+	+	+	-
CLC	/	/	/	/
SBC	+	+	+	-
SEC	/	/	/	/
BEQ	/	/	/	/
BCC	/	/	/	/
BCS	/	/	/	/
BMI	/	/	/	/
BPL	/	/	/	/
BVC	/	/	/	/
BVS	/	/	/	/
CMP	+	+	+	-
CPX	-	-	-	-
CPY	-	-	-	-
BIT	-	-	-	-
CLV	/	/	/	/
NOP	/	/	/	/
TAX	/	/	/	/
TAY	/	/	/	/
TXA	/	/	/	/
TYA	/	/	/	/
JMP	-	-	-	-
JSR	-	-	-	-

+ anwendbar
– nicht erlaubt
/ weder absolute noch Zeropage-Adressierung möglich

Tabelle 5. Anwendbarkeit der indizierten Adressierungsarten auf die bisher gelernten Assembler-Befehle.

Absolut-X-indiziert zum Beispiel LDA 1500,X
Absolut-Y-indiziert zum Beispiel LDA 1500,Y
Zero-page-absolut-X-indiziert zum Beispiel LDA 2B,X
Zero-page-absolut-Y-indiziert zum Beispiel LDX 2B,Y

Die Verwendung des Y-Registers als Indexregister ist stark eingeschränkt. Nur bei wenigen Befehlen ist sie erlaubt (tatsächlich nur LDX und STX bei Zero-page-absolut-indizierter Adressierung). In der Tabelle 5 sehen Sie, welche bisher behandelten Befehle wie mit der indizierten Adressierung verwendet werden dürfen.

Es gibt noch zwei weitere Arten einer indizierten Adressierung, auf die wir noch zu sprechen kommen werden.

27. Einige Nachzügler: Die Befehle BIT, CLV, NOP und TAX, TAY, TXA, TYA

Wir wollen noch ein bißchen aufräumen: Ein paar Befehle, die bisher zu keinem Gebiet so richtig paßten, sollen jetzt behandelt werden.

BIT: Dieser Befehl heißt »Bit-Test» und paßt von daher eigentlich zu den in Kap. 23 behandelten Vergleichsbefehlen. Die Behandlung der Flaggen ist aber völlig anders. Nehmen wir das Beispiel

BIT 1500

Folgendes passiert: Der Inhalt der Speicherstelle $1500 wird mit dem Inhalt des Akku UND-verknüpft, das Ergebnis in der Z-Flagge angezeigt und Bit 7 sowie Bit 6 von $1500 in die N- beziehungsweise die V-Flagge übertragen. Weder Akku noch Inhalt von $1500 verändern sich dabei.

Das ging ein bißchen holterdipolter. Sehen wir uns das jetzt mal ganz langsam Schritt für Schritt an! Zunächst die UND-Verknüpfung. Bit für Bit wird der Akku-Inhalt mit dem Inhalt der adressierten Speicherstelle UND-verknüpft. Dabei gelten folgende Regeln (die Leser der Grafik-Serie kennen das ja schon):

0 UND 0 = 0
0 UND 1 = 0
1 UND 0 = 0
1 UND 1 = 1

Nur dann also, wenn die entsprechenden Bits im Akku und in 1500 gleich 1 sind, ergibt sich bei der UND-Verknüpfung eine 1. Man stellt sowas meist in einer sogenannten Wahrheitstabelle zusammen (Tabelle 6).

UND	0	1
0	0	0
1	0	1

Tabelle 6. Wahrheitstabelle der logischen Verknüpfung UND

Nehmen wir als Beispiel mal an, im Akku stünde $0A und in der Speicherstelle $1500 wäre $09 enthalten. Die UND-Verknüpfung sieht dann so aus:

```
Akku  $0A 0000 1010
1500  $09 0000 1001
UND   _____________
          0000 1000
```

Das Ergebnis ist also $08. In der Z-Flagge wird in dem Fall, daß das Ergebnis der UND-Verknüpfung ungleich Null ist (wie hier) eine Null angezeigt, sonst eine 1.

Wir haben in unserem Zahlenbeispiel mit dem BIT-Befehl überprüft, ob die Bits 1 und 3 in Speicherstelle $1500 gelöscht sind. Dazu haben wir in den Akku eine sogenannte Maske (hier also $0A) geladen. Das Ergebnis sagt uns, daß nicht beide Bits gelöscht waren. Wäre der Inhalt von $1500 beispielsweise $10 gewesen (0001 0010), hätten wir in der Z-Flagge eine 1 gefunden. Daher der Name »Bit-Test«: Durch geeignete Maskenwahl kann praktisch jedes Bit überprüft werden. Dabei werden weder der Akku-Inhalt noch der Inhalt der angesprochenen Speicherstelle verändert.

Der BIT-Befehl hat aber noch mehr Auswirkungen: Die Bits 6 und 7 der geprüften Speicherzelle findet man nach Befehlsausführung in zwei Flaggen nochmal:

Bit 7 in der N-Flagge
Bit 6 in der V-Flagge

Damit kann man beispielsweise überprüfen, ob sich am adressierten Ort eine negative Zahl befindet. Alle drei Flaggen können ja nun mit den Branch-Befehlen abgefragt werden. Sie erkennen sicherlich schon, wie vielseitig dieser merkwürdige BIT-Befehl einsetzbar ist.

Adressierbar ist BIT entweder absolut (wie im obigen Beispiel) oder Zeropage-absolut. Je nachdem liegt er dann als 3-Byte- oder als 2-Byte-Befehl vor.

CLV: Dieser Befehl heißt »Clear oVerflow-flag«, also »lösche die Überlauf-Flagge«. Die V-Flagge war –wie Sie sich erinnern werden –unsere rote Ampel bei Rechenoperationen (siehe Kap. 16). Es ist ein 1-Byte-Befehl mit impliziter Adressierung und interessant daran ist, daß es keinen Befehl gibt, der das Gegenteil –also das Setzen der V-Flagge –bewirkt.

NOP: NOP steht für »No OPeration«, was bedeutet »keine Tätigkeit«. Das ist der Nichtstu-Befehl. Er tut aber doch etwas: Er sorgt dafür, daß der Befehlszähler weitergezählt wird und bewirkt eine Verzögerung von 2 Taktzyklen. NOP ist ein 1-Byte-Befehl mit impliziter Adressierung. Er wird in fertigen Programmen nur selten verwendet: Zur Erzeugung einer kurzen definierten Verzögerung. Meist gebraucht man ihn bei der Erstellung eines Programmes als Platzhalter oder bei der Fehlersuche, um zum Beispiel unerwünschte Sprünge zu ersetzen.

Die Transporteure: TAX, TAY, TXA und TYA

Ab und zu ist es nötig, Registerinhalte untereinander auszutauschen. Viele Dinge (Addition, Subtraktion und so weiter) können nur im Akku geschehen. Wenn wir eine solche Operation beispielsweise mit dem Inhalt des X-Registers ausführen wollen, verschieben wir diesen Inhalt mit dem Befehl TXA. »Transfer X into Accumulator« also »übertrage X-Register in den Akku« bedeutet das. Analog verwendet man TYA, um Y-Register-Inhalte in den Akku zu schieben oder für den umgekehrten Weg TAY beziehungsweise TAX (Akkuinhalt ins Y- beziehungsweise ins X-Register schieben). Genau genommen wird nicht übertragen, sondern nur kopiert: Die Register, aus denen verschoben wird, bleiben unverändert. Weil die jeweiligen Zielorte der Verschiebung (Akku, X- oder Y-Register) vom neuen Inhalt überschrieben werden, können sich auch Flaggen ändern. Betroffen sind von dieser Möglichkeit die N- und die Z-Flagge. Alle vier Befehle bestehen aus einem Byte und können natürlich nur implizit adressiert werden.

28. So springen die Assembler-Alchimisten: JMP, JSR

JMP und JSR entsprechen ungefähr den vom Basic her bekannten Befehlen GOTO und GOSUB.

JMP kommt von »JuMP to address«, also »springe zur angegebenen Adresse«. Nehmen wir uns wieder ein Beispiel vor:

JMP 1500

bewirkt einen Sprung zur Adresse 1500. Das funktioniert so: In den Programmzähler werden LSB und MSB der Ziel-

adresse geladen. Das war dann auch schon der Sprung, denn der Programmzähler ist der Pfadfinder des Computers: Die Adresse, die dort steht, wird als nächste bearbeitet. Schalten Sie doch mal den SMON ein (oder einen anderen Monitor) und sehen Sie sich das mit folgenden Befehlen an:

```
1400   JMP   1500
```

Dort unterbrechen wir den Computer mit

```
1500   BRK
```

So weit, so gut: Wir starten mit dem SMON-Kommando G 1400 und erhalten eine Registeranzeige mit dem Programmzählerstand 1501. Genau das hatten wir ja erwartet.
Weniger durchschaubar ist das folgende Beispiel:

```
1400   LDA   #00
1402   LDX   #16
1404   STA   1300
1407   STX   1301
140A   JMP   (1300)
```

Dazu gehört dann noch die Programmzeile:

```
1600   BRK
```

Wenn Sie das genauso eingegeben haben und dann mittels G 1400 starten, erhalten Sie eine Registeranzeige mit dem Programmzählerstand 1601.

Schon an der neuen Schreibweise des Argumentes in Zeile 140A werden Sie bemerkt haben, daß hier nicht mehr die normale absolute Adressierung wie zuvor angewendet wird. Dies ist eine neue Form: Die **indirekte Adressierung**. Indirekt deswegen, weil wir nicht mehr direkt die Zieladresse angeben, sondern einen sogenannten Vektor. Ein Vektor besteht aus zwei aufeinander folgenden Speicherzellen (hier also 1300 und 1301), die in der Form LSB/MSB die eigentliche Zieladresse enthalten. Das LSB von $1600 ist $00. Das haben wir über den Akku nach $1300 geladen. Das MSB $16 kam durch das X-Register an seinen Platz $1301:

```
Zieladresse  16     00
             MSB    LSB
             ↑      ↑
Vektor       1301   1300
```

Das ist die Methode der toten Briefkästen, die in Kreisen der Assembler-Alchimisten anscheinend genauso beliebt ist wie bei Agenten. So wie diese im hohlen Baum die Treffpunktanschrift hinterlegt finden, verläßt sich unser Computer auf die Speicherstellen 1300 und 1301 für die Angabe der Zieladresse.

Diese Art der Adressierung ist im wahrsten Sinn des Wortes ein Unikum: Es gibt sie nämlich nur für den JMP-Befehl! Davon wird allerdings dann auch recht häufig Gebrauch gemacht, zum Beispiel im Betriebssystem unseres Computers. Aber darüber und über die Vektoren, die dazu verwendet werden, soll ein andermal berichtet werden.

Wir dürfen nämlich nicht den anderen Sprungbefehl JSR vergessen. JSR steht für »Jump to SubRoutine«, was eingedeutscht etwa bedeutet »springe zum Unterprogramm«. Genauso wie in Basic Unterprogramme durch GOSUB Zeilennummer aufgerufen werden, kann das auch hier geschehen durch JSR Adresse. Hier ist nur die absolute Adressierung möglich. Das erste Beispiel soll uns zeigen, wie dieser Befehl funktioniert:

```
1400   JSR   1500
```

Dort soll dann erstmal stehen:

```
1500   BRK
```

Noch nicht starten!! Zunächst einmal verzeihen Sie mir diese Programmierer-Todsünde: Aus einem Unterprogramm heraus den Programmablauf zu beenden! Ich werd's auch nie wieder tun. Hier geschieht das nur zu Lehrzwecken. Was läuft ab: Der Programmzählerinhalt plus 2 wird auf den Stapel gelegt und dann die Adressse 1500 in den Programmzähler geladen. Ebenso kurz wie unklar! Was ist denn ein Stapel? Also langsam, Schritt für Schritt.

Der Sinn von Unterprogrammen ist ja, daß der Computer nach Ende der Bearbeitung wieder ins aufrufende Hauptprogramm zurückkehrt. Er muß sich aber dazu irgendwo merken, von wo aus er zum Unterprogramm gesprungen ist. Dazu verwendet er den Stapel. Das ist ein Speicherbereich ($0100 bis $01FF), der direkt vom Prozessor aus verwaltet wird. Die genaue Architektur und Handhabung dieses »Prozessor-Stack« werden wir noch in einer späteren Folge kennenlernen. Uns soll hier nur interessieren, daß es einen Zeiger gibt, der auf den nächsten freien Platz im Stapel weist und daß dieser Speicher von oben nach unten gefüllt wird (wie in Basic bei den Strings). Wenn Sie mit Hilfe des SMON mal in den Stapel hineinsehen wollen, dann geben Sie doch mal ein M 0100 01FF. Was nun genau bei Ihnen drin steht, ist sehr von der vorherigen Nutzung Ihres Computers abhängig. Der Mikroprozessor nutzt den Stapel bei sehr vielen Tätigkeiten. Es kommt auch nur auf den Teil des Stapels an, der durch den Stapelzeiger als gefüllt bezeichnet wird. Der Stapelzeiger wird beim SMON in der Registeranzeige als SP angezeigt. Wenn Ihr Stapelzeiger (prüfen Sie das doch mal durch Eingabe von R) nun zum Beispiel F6 zeigt, dann bedeutet das, daß alle Stapelplätze von $01F6 an abwärts frei und die oberhalb bis $01FF besetzt sind. Beim Nachsehen mit M 01F0 01FF finden Sie dann beispielsweise:

```
:01F0  20  00  20  AA  C1  FA  C0  46
                               --
:01F8  E1  E9  A7  A7  79  A6  9C  E3
```

Die Speicherstelle, auf die der Stapelzeiger weist, ist unterstrichen. Nun starten wir mit G 1400 unser kleines verbotenes Testprogramm. Es meldet sich die Registeranzeige. Im Stapelzeiger steht jetzt F4 (oder eben Ihr vorhergegangener SP minus 2). Wenn wir nun wieder im Stapel nachsehen mit M 01FO 01FF, dann finden wir im Gegensatz zur obigen Anzeige nun:

```
:01F0  20  AA  C1  FA  C0  02  14  46
                       --  ↑↑  ↑↑
:01F8  E1  E9  A7  A7  79  A6  9C  E3
```

Unterstrichen ist wieder das Ziel des Stapelzeigers, der jetzt zwei Plätze weitergerückt ist, um der durch Pfeile gekennzeichneten Adresse 1402 (als LSB/MSB) Raum zu schaffen. $1402 ist das letzte Byte des JSR-Befehls. Wie wir den Programmzähler kennen, ist er im allgemeinen immer einen Schritt voraus. Hier liegt er aber einen zurück, falls er nach Beendigung des Unterprogrammes an der notierten Adresse weitermacht. Dazu kommen wir gleich noch. Was wir am Programmzähler aber auch noch nach Ablauf unseres kurzen Beispielprogrammes ablesen können, ist die Tatsache, daß die Sprungadresse 1500 in ihn geschrieben wird, somit der Sprung dann also stattgefunden hat.
Nun bauen wir das kleine Programm etwas um:

```
1400   JSR   1500
1403   BRK
```

Das Unterprogramm soll nur aus dem Rücksprung bestehen:

```
1500   RTS
```

Verlangen Sie nun noch vor dem Start eine Registeranzeige mit R und merken Sie sich den Wert des Stapelzeigers. Dann starten Sie das Programm mit G 1400 und achten Sie auf die neue Registeranzeige. Zwei Dinge interessieren uns:
1) Der Wert des Stapelzeigers ist unverändert geblieben.
2) Der Programmzähler weist nun auf $1404.

Wenn Sie nun nochmal mit dem M-Befehl des SMON in den Stapel sehen, werden Sie unter Umständen zwar noch die Adresse 1402 dort finden (dann nämlich, wenn wir den Stapel seit dem letzten Programm nicht verändert haben). Wie Sie aber inzwischen wissen, hätte durch den neuen JSR-Befehl nochmal 1402 dort eingetragen sein müssen. Das stand da auch einige Mikrosekunden lang... bis der RTS-Befehl wirksam wurde. RTS macht ziemlich viel:
1) RTS holt die auf dem Stapel gespeicherte Adresse ab, und schreibt sie in den Programmzähler.

2) RTS vermindert dabei den Stapelzeiger um 2.
3) RTS addiert zum Programmzähler eine 1.

Deswegen kann das Programm also bei $1403 weiterlaufen und der Programmzähler nun hinter dem BRK-Befehl stehen.

Machen Sie doch mal etwas anscheinend total Verrücktes: Starten Sie mit G 1500. Es gibt da zwei Möglichkeiten, was geschehen kann: Entweder stand da noch vom ersten unterbrochenen Testprogramm die Adresse 1402. Dann endete nun alles mit einer Registeranzeige, bei der der Stapelzeiger um 2 höher gerutscht ist.

Oder da stand diese Adresse nicht mehr. Dann befinden Sie sich nun wieder im Basic. Wieso eigentlich? Als nächste Adresse finden Sie auf dem Stapel $E146 (dez.57670). Diese Adresse + 1 wird ja durch RTS in den Programmzähler gerufen. Ein Sprung an diese Adresse ist ein Sprung in ein Programm des Betriebssystems. Haben Sie ein ROM-Listing? Dann sehen Sie mal nach: Dort steht der Befehl...RTS. Dies neuerliche RTS holt nun jedenfalls die nächste Adresse vom Stapel: $A7E9 (dez.42985). Diese Adresse + 1 im Programmzähler führt unseren Computer in die Basic-Interpreter-Schleife, also ins Basic zurück.

Wir haben so viel über den Stapel gehört, daß wir JSR fast schon wieder aus den Augen verloren haben. Deswegen nochmal eine kurze Übersicht:

a) JSR speichert den Programmzählerwert des letzten Bytes des Befehls auf dem Stapel zum Beispiel 1402,
b) stellt dabei den Stapelzähler um 2 zurück zum Beispiel von $F6 nach $F4
c) schreibt in den Programmzähler die angegebene Zieladresse, zum Beispiel 1500
d) Das Unterprogramm wird abgearbeitet bis der RTS-Befehl auftaucht.
e) Dann wird die gemerkte Adresse +1 in den Programmzähler geschrieben, zum Beispiel 1402+1=1403
f) und dabei der Stapelzähler wieder um 2 erhöht, zum Beispiel von $F4 wieder zu $F6
g) Das Programm läuft nun wieder nach dem JSR-Befehl weiter, zum Beispiel also bei 1403.

Nun sollte eigentlich auch klar sein, warum ein Aussprung aus einem Unterprogramm oder ein Abbruch im Unterprogramm eine Programmierer-Todsünde ist: Der Stapelzeiger wird nicht zurückgestellt. Die gemerkte Rücksprungadresse versauert allmählich auf dem Stapel. Noch schlimmer sind solche Sachen in einer Schleife, wo mehrfach aus dem Unterprogramm ausgebrochen wird: Hier ist der Stapel bald voll Müll und der Computer beendet seine Zusammenarbeit mit dem Programmierer. Weil aber Basic-Programme nichts anderes sind als eine Folge von Maschinenprogrammen, die je nach Befehl durch den Interpreter aneinandergereiht werden, ist das auch in Basic eine Todsünde. Wir wollen aber nicht so hart mit uns umgehen: Wenn wir gelernt haben, wie man mit speziellen Assembler-Befehlen im Stapel herumschaufeln kann, dann haben wir bei richtiger Anwendung von vorneherein jedenfalls in diesem Punkt die Absolution erhalten.

29. Alles fließt: Fließkommazahlen

Jeder, der tiefer in die Geheimnisse der Assembler-Alchimie eindringen will, muß sich vertraut machen mit der häufigsten Art der Zahlenverarbeitung in unserem Computer. Das ist die Handhabung von Fließkommazahlen (auch Gleitkommazahlen genannt). Wir werden dazu folgende Fragen zu klären haben:

1) Was sind Fließkommazahlen?
2) Wie sehen sie im binären Zahlensystem aus?
3) Wie behandelt unser Computer positive und negative Fließkommazahlen?
4) Wie können wir als Programmierer Einfluß nehmen auf die Verarbeitung dieser Zahlen im Computer?

Die Behandlung dieser vier Fragen wird uns eine ganze Weile beschäftigen. Fangen wir mit der ersten an: In Standardwerken der Mathematik werden Sie lange suchen müssen, um den Begriff »Fließkommazahl« zu finden. Im deutschen Sprachraum gibt es häufiger die Bezeichnung »wissenschaftliche Zahlendarstellung«. Das klingt sehr hochgestochen und ist eigentlich ganz einfach. Leser der Grafik-Serie werden sich vielleicht noch erinnern: Die Zahl 1000 kann man auf verschiedene Weise darstellen:

1000 = 10 * 10 * 10 = 10^3 (in Basic 10↑3)

Die hochgestellte Zahl (in Computerschreibweise: Die Zahl hinter dem Hochpfeil) ist hier gleich der Anzahl der Stellen minus 1 (1000 hat vier Stellen, also ist die Hochzahl eine 3). Diese Hochzahl nennt man Exponent (vom lateinischen exponere = anzeigen, herausheben). Nehmen wir nun einige andere Zahlen:

200 = 2 * 100 = 2 * 10↑2

oder

2500 = 2,5 * 1000 = 2,5 * 10↑3

Ich glaube, jetzt beginnt es Ihnen klarzuwerden, daß man auf diese Art wohl alle Zahlen irgendwie darstellen kann. Man dröselt die Zahlen auseinander, bildet ein Produkt, von dem der eine Multiplikator durch 10 teilbar ist (durch die Basis unseres normalen Zahlensystems). Genauer gesagt: Ein Faktor (also in den Beispielen 1000 oder 100) ist darstellbar als Potenz von 10. Der andere Faktor (in den Beispielen 1 oder 2 oder 2,5) wird Mantisse (vom lateinischen manitissa = Zugabe, Anhang, Schleppe) genannt. Sehen wir uns nochmal 2500 an:

```
2500 =   2,5 * 1000 =    2,5 * 10↑3
     =    25 *  100 =     25 * 10↑2
     =   250 *   10 =    250 * 10↑1
     =  2500 *    1 =   2500 * 10↑0
```


Das letzte war nur der Vollständigkeit halber, denn irgendeine Zahl hoch 0 ist immer 1. Man kann auch aus der 2500 folgendes machen:

```
2500 = 0,25  *  10000 = 0,25  * 10↑4
oder = 0,025 * 100000 = 0,025 * 10↑5
```

und so weiter. Oder anders herum:

```
2500 =  25000 * 0,1  =  25000 * 10↑-1
     = 250000 * 0,01 = 250000 * 10↑-2
```

und so weiter.

Dabei bedeutet:

$10^{-2} = 1/10^{-2} = 0,01$

Man kann sich das merken, indem man die Anzahl der Stellen zählt, um die man das Komma verschiebt. Diese Anzahl addiert man dann zur Hochzahl. Zur Erläuterung:

$0,12345 = 1,2345 * 10^{-1}$

Wir haben das Komma um eine Stelle nach rechts gerückt, weshalb wir die Hochzahl -1 schreiben müssen (vorher war da nämlich unsichtbar die Hochzahl 0: und 10↑0=1).

$0,12345 = 123,45 * 10^{-3}$

Hier wurde das Komma um drei Stellen nach rechts verschoben. Daher der Exponent -3. Sie sehen folgenden Zusammenhang:

Komma eine Stelle nach rechts verschoben: Exponent + (-1).

Zum Beispiel

$0,1234 * 10^{-2} = 1,234 * 10^{-3}$

Komma eine Stelle nach links verschoben: Exponent +1.

Zum Beispiel

$3,14 * 10^{-2} = 0,314 * 10^{-3}$

Verstehen Sie nun, warum man diese Art der Zahlendarstellung Fließkomma- oder Gleitkommazahlen nennt?

Vielleicht sehen Sie aber noch nicht den Sinn der Fließkommazahlen ein. Dazu gebe ich Ihnen zwei einsichtige Bei-

spiele. Der Atomkern eines Heliumatoms wiegt etwa (halten Sie sich fest):
0,000 000 000 000 000 000 000 000 006 643 kg.

Sehr unbequem, diese ganzen Nullen immer mitzuschleppen. Wir verschieben deshalb das Komma um 27 Stellen nach rechts und schreiben dann

$6{,}643 * 10^{-27}$ kg.

2. Beispiel: Wir haben einen Ballon mit diesem Gas gefüllt. Bei normalen Temperatur- und Luftdruckbedingungen befinden sich in einem Kubikzentimeter im Ballon ungefähr (nochmal festhalten!):
26 900 000 000 000 000 000 Heliumatome

Wieder eine recht unangenehme Nullschlepperei. Wir verschieben das Komma um 19 Stellen nach links und erhalten $2{,}69 * 10^{-19}$ Heliumatome. Fein, nicht wahr!

Abgesehen von der höheren Bequemlichkeit: Der Computer müßte allerhand Speicherplatz zur Handhabung der vielen Nullen bereitstellen. Mit BCD-Zahlen könnten wir zwar jede Zahl erfassen, hätten aber immer unterschiedlich viele Bytes zu verarbeiten. Wenn wir Fließkommazahlen verwenden, können wir – wie Sie noch sehen werden – jede (na sagen wir mal: fast jede) Zahl in der gleichen Anzahl Bytes aufbewahren.

Vom Basic her kennen Sie Fließkommazahlen auch (hier wird das Komma allerdings durch den Punkt ersetzt, entsprechend der angloamerikanischen Schreibweise). Das sind die, wo man zum Beispiel schreibt 6.02E23 oder 6.02E+23, was dann bedeutet $6{,}02 * 10^{-23}$. E steht dort für Zehnerexponent. Durch die Art, wie Fließkommazahlen im normalen Computerdasein gespeichert werden, ergeben sich obere und untere Grenzen. Die höchste in Basic verarbeitbare Zahl im C 64 ist

$+1.70141183 * 10^{-38}$

Größere Zahlen verursachen in Basic einen OVERFLOW ERROR. Was in Maschinensprache mit größeren Zahlen geschieht, ist weitgehend unsere Sache. Die dem Betrag nach kleinste verarbeitbare Zahl ist

$\pm 2.93873588 * 10^{-39}$

In Basic arbeitet bei Unterschreitung der Computer einfach mit einer Null weiter. Für die Behandlung in Maschinensprache sind ebenfalls wir als Programmierer verantwortlich.

Für diesmal sei's genug der Zahlenspiele: Später werden wir uns weiter mit Fließkommazahlen befassen.

30. Die USR-Funktion

Wieder einmal soll uns das Zusammenspiel von Basic und Maschinensprache beschäftigen. Einen Aufruf von Maschinenroutinen – nämlich den mit SYS – haben wir schon kennengelernt. Wir POKEten die zu übergebenden Werte an die Abrufspeicherstellen. Bei diesen Werten hat es sich um einfache Integerzahlen gehandelt, zum Beispiel die Anzahl der Glieder einer zu summierenden arithmetischen Reihe. Was tun wir aber, wenn wir Fließkommavariable an ein Maschinenprogramm übermitteln wollen? Gewiß, werden Sie sagen, lernen wir das ja noch und können dann entsprechende POKE-Kommandos geben. Damit haben Sie auch recht, nur ist das dann der »harte« Weg. Es gibt auch einen problemlosen »weichen« Weg, nämlich das USR-Kommando.

USR ist ein Basic-Befehl und rührt her von »USeR callable machine language subroutine«, also »durch den Benutzer aufrufbares Maschinensprachunterprogramm«. Darin liegt eigentlich noch nichts Neues gegenüber dem SYS-Befehl. Im Gegensatz zu SYS – wo das Argument die Einsprungadresse des Maschinenprogrammes ist – übergibt USR als Argument eine beliebige Fließkommavariable in festgelegter Form an eine sehr nützliche Speicherstellenkombination, den Fließkomma-Akkumulator 1, von uns künftig einfach FAC genannt. Der FAC belegt die Speicherstellen 97 bis 102 ($61 bis $66). Wenn das eventuell in Basic benötigte Ergebnis dort auch in der vorgeschriebenen Form abgelegt wird, kann es im Basic-Programm weiterverwendet werden. Keine Angst, dazu kommen wir bei der weiteren Behandlung der Fließkommazahlen noch ganz ausführlich zu sprechen. Heute soll uns das noch nicht belasten. Als Argument kann man nämlich auch irgendeine bedeutungslose Größe, ein sogenanntes Dummy angeben, das dann gar nicht weiter verwendet wird. Der USR-Befehl dient in diesem Fall lediglich dem bequemen Ansteuern eines Maschinenprogrammes.

Woher weiß unser Computer beim USR-Befehl, welche Maschinenroutine er im 64-KByte-Speicher bearbeiten soll? Beim SYS-Befehl ist das klar: Das Argument sagt es:

```
SYS 24345
```

läßt den Programmzähler auf dez.24345 zeigen. Aber wenn wir eingeben:

```
USR(24345)
```

dann packt der Computer die Zahl 24345 als Fließkommavariable in den FAC und meldet dann einen SYNTAX ERROR. Das liegt daran, daß der Basic-Interpreter beim USR-Befehl einen der oben kennengelernten indirekten Sprünge vollführt:

```
JMP     (311)
```

$311/312 (in dezimal 785/786) ist also ein Vektor, und der weist im Normalfall zu einer Routine, die den SYNTAX ERROR ausgibt (dez. 45640). Bevor wir also den USR-Befehl geben, müssen wir in diesen Vektor die Startadresse unserer Maschinenroutine schreiben:

```
dez. 24345 = $5F19
LSB $19 = dez. 25 in Speicher 785 mit
                POKE 785,25
MSB $5F = dez. 95 in Speicher 786 mit
                POKE 786,95
```

Jetzt weiß der Computer, wohin er beim USR-Aufruf springen soll, und solange, bis wir den Vektor wieder ändern, führt er bei jedem USR-Befehl unser bei 24345 stehendes Maschinenprogramm aus. Wir müssen nur noch dafür sorgen, daß dort dann auch wirklich eines anfängt. Ein Beispiel werden wir nachher noch behandeln.

31. Der harte Kern: Nochmal Speicherfragen

Die Struktur des C 64-Speichers ist vereinfacht schon in der Grafik-Serie und zu Beginn dieses Kurses gezeigt worden. Dabei tauchten zwei ROM-Bereiche auf, die wir Basic-Interpreter und Betriebssystem genannt haben. Diese Unterteilung ist nicht ganz korrekt. Wenn Sie über ein ROM-Listing verfügen und beispielsweise das Ende des ROM-Bereiches von $A000 bis $BFFF sowie den Anfang des oberen ROM ($E000 bis $FFFF) untersuchen, dann stellen Sie fest, daß ab dez. 49087 ($BFBF) die Basic-Funktion EXP bearbeitet wird. Der letzte Befehl vor $C000 beendet diese Funktion aber nicht etwa, sondern dort steht:

```
JMP     E000
```

Tatsächlich läuft ab $E000 bis $E042 die Bearbeitung der EXP-Funktion munter weiter, und auch danach finden sich allerlei Basic-Befehle (SIN, COS und so weiter). Da liegt also keine klare Trennung vor, sondern ein Mischmasch. Wir sollten uns vielleicht angewöhnen – statt vom Interpreter und dem Betriebssystem –, vom unteren und oberen ROM-Bereich zu sprechen.

Eine andere Unterscheidung ist dagegen sinnvoll: Wie einige Besitzer neuerer Commodore 64 sicherlich bemerkt

haben, sind Teile der ROM-Routinen im Laufe der Zeit verändert worden. Hauptsächlich geht es bei den aktuellen Neuerungen dieser internen Maschinenprogramme um die Farbgebung der Zeichen. Man kann eigentlich nie so recht wissen, was den Software-Planern von Commodore noch alles einfällt. Jedenfalls können deren Ideen manchmal recht dramatische Folgen haben, nämlich dann, wenn Sie ein fabelhaftes Maschinenprogramm gebaut haben, welches ROM-Routinen direkt verwendet. Der Programmierer spielt auf diese Weise eine milde Form des russischen Roulettes. Glücklicherweise halten sich die Änderungen in Grenzen, und wir dokumentieren unsere Programme ja auch immer gut (Sie etwa nicht??). Notwendige Umbauten können also leicht vonstatten gehen.

Ganz ohne ROM-Routinen-Verwendung kommt man eigentlich kaum aus. Es gibt aber einen ROM-Bereich, für den Commodore verspricht, keinerlei Änderungen durchzuführen: die Kernel-Sprungtabelle.

Das ist ein Programmbereich ($FF81 bis $FFF5), in dem 39 JMP-Befehle enthalten sind (zum Teil in absoluter, aber auch in indirekter Adressierung). Jeder dieser Sprungbefehle weist auf die Einsprungadresse eines Maschinenprogrammes. Da finden sich alle wichtigen Ein/Ausgabe-Operationen, Systemtakt- und Uhrsteuerungen und anderes mehr. Wir werden uns nach und nach damit vertraut machen. In der Tabelle 7 sind die Kernel-Adressen und ihre Funktion aufgeführt. Manche davon können ohne jede Vorbereitung benutzt werden, andere brauchen bestimmte Routinen oder Angaben, um sinnvoll zu arbeiten.

Die Absicht von Commodore ist es, daß jeder Aufruf von zum Beispiel $FFD2 die Ausgabe eines Zeichens bewirkt, und zwar unabhängig davon, welchen Computer in welcher Version wir benutzen. Das Programm, welches diese Zeichenausgabe letztendlich ausführt, kann sich ändern, kann in ganz andere Speicherbereiche gelegt werden. An der Stelle $FFD2 wird aber immer ein JMP mit der Einsprungadresse stehen. Leider ist diese Sprungtabelle viel zu knapp gehalten. Es gibt so viele interessante ROM-Routinen, die wir alle ohne diese schöne Sicherheit anspringen müssen.

Adresse HEX	Adresse dezimal	Name	Funktion
FF81	65409	CINT	Prüfen der TV-Norm, Berechnung der Taktfrequenz
FF84	65412	IOINIT	Ein/Ausgabe-Reset
FF87	65415	RAMTAS	Prüfen auf freien Basic-RAM
FF8A	65418	RESTOR	Initialisieren der I/O-Vektoren
FF8D	65412	VECTOR	Lesen und Setzen der I/O-Vektoren
FF90	65424	SETMSG	Setzen des Ausgabe-Modus
FF93	65427	SECOND	Ausgeben der Sekundäradresse nach LISTEN
FF96	65430	TKSA	Ausgabe der Sekundäradresse nach TALK
FF99	65433	MEMTOP	Lesen/Setzen des Speicherendes
FF9C	65436	MEMBOT	Lesen/Setzen des Speicheranfangs
FF9F	65439	SCNKEY	Abfragen der Tastatur
FFA2	65442	SETTMO	Setzen der Time-Out-Flagge
FFA5	65445	ACPTR	Zeichen vom seriellen Port in Akku lesen
FFA8	65448	CIOUT	Zeichen vom Akku auf seriellen Port ausgeben
FFAB	65451	UNTLK	Sendet UNTALK an seriellen Bus
FFAE	65454	UNLSN	Sendet UNLISTEN an seriellen Bus
FFB1	65457	LISTEN	Sendet LISTEN an Geräte per seriellen Bus
FFB4	65460	TALK	Sendet TALK an Geräte per seriellen Bus
FFB7	65463	READST	Liest I/O-Status in den Akku
FFBA	65466	SETLFS	Festlegung der Parameter für OPEN
FFBD	65469	SETNAM	Festlegung des Filenamens
FFC0	65472	OPEN	Öffnet spezifizierten File
FFC3	65475	CLOSE	Schließt spezifizierten File
FFC6	65478	CHKIN	Öffnet einen Eingabekanal
FFC9	65481	CHKOUT	Öffnet einen Ausgabekanal
FFCC	65484	CLRCHN	Schließt Ein- und Ausgabekanäle
FFCF	65487	CHRIN	Holt vom aktiven Eingabekanal ein Zeichen in den Akku
FFD2	65490	CHROUT	Sendet Akku-Inhalt auf aktiven Ausgabekanal
FFD5	65493	LOAD	LOAD und VERIFY von Programmen
FFD8	65496	SAVE	Speichern von Programmen
FFDB	65499	SETTIM	Uhrzeit setzen
FFDE	65502	RDTIM	Uhrzeit lesen
FFE1	65505	STOP	STOP-Taste abfragen
FFE4	65508	GETIN	Zeichen aus dem Tastaturpuffer in den Akku lesen
FFE7	65511	CLALL	Schließen aller Kanäle und Files
FFEA	65514	UDTIM	Uhr um 1/60 Sekunde weiterzählen
FFED	65517	SCREEN	Lesen des Bildschirmformates
FFF0	65520	PLOT	Lesen/Setzen der Cursor-Position
FFF3	65523	IOBASE	Lesen der Startadresse der Ein- und Ausgabebausteine

Tabelle 7. Kernel-Routinen

32. Die Urzelle eines Programmprojektes

Wir sind jetzt soweit, daß wir die Urzelle eines Programmprojektes, welches uns eine lange Zeit begleiten wird, aufbauen können. Wir wollen etwas unter den Teppich kehren. Der Teppich, das sind die uns bislang nicht zugängigen RAM-Bereiche unter den ROMs. Haben Sie das nicht auch schon mal erlebt, daß Sie während einer Programmarbeit plötzlich feststellen, Sie benötigen zum Beispiel für eine Zwischenrechnung ein weiteres Programm, oder Sie wälzen Listen und denken sich, ein kleiner Hilfsbildschirm wäre jetzt von Nutzen, oder....

Mit diesem heute zu startenden Programm wäre all das und noch viel mehr realisierbar. Es soll auf einfache Weise beliebige Speicherbereiche unters ROM schieben und sie wieder hervorholen können.

Natürlich braucht die Entwicklung dieses Projektes einige Zeit, zumal wir noch vieles lernen müssen. Deswegen sind wir

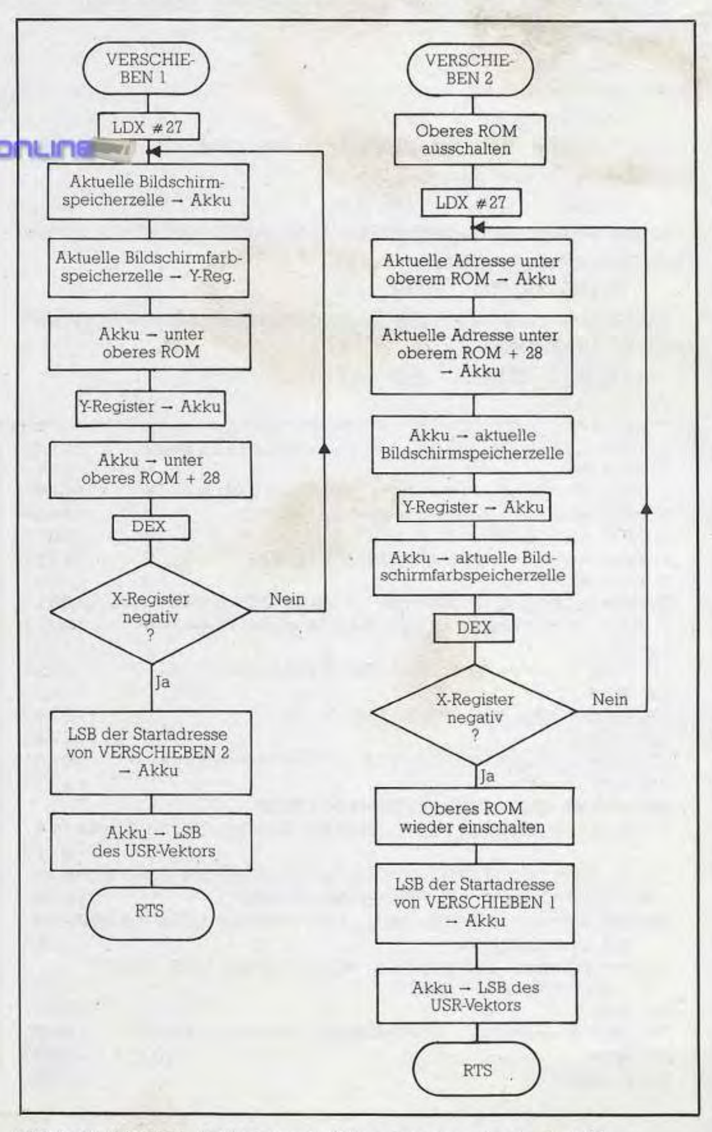


Bild 17. Das Flußdiagramm zu dem im Text erklärten Programm.

in dieser ersten Urzelle noch sehr eingeschränkt: Wir verschieben zuerst einmal nur eine Bildschirm-Kopfzeile unter den oberen ROM-Bereich. Auch in dieser einfachsten Version gibt es noch einige Programmteile, die Sie erst nach der nächsten Ausgabe verstehen werden. Aber irgendwann müssen wir ja mal anfangen, Nägel mit Köpfen zu machen.

Unser Maschinenprogramm soll durch die USR-Funktion aufgerufen werden. Wie wir es in dieser Ausgabe gelernt haben, muß deshalb vor dem ersten Aufruf eine Initialisierung durch Belegen des USR-Vektors mit unserer Startadresse stattfinden. Die Startadresse soll $02B6 (dez. 694) sein, denn dort gibt es einen freien RAM-Bereich bis inklusive $02FF (dez. 767), der weder andere Programme noch Kassettenoperationen stört. Das MSB $02 ist dezimal auch 2 und wird nach 786 gePOKEt:

```
POKE 786,2
```

Das LSB $B6 ist dezimal 182 und soll in 785 geschrieben werden:

```
POKE 785,182
```

Damit ist der USR-Vektor gestellt und wir brauchen uns nicht mehr weiter darum zu kümmern: Jeder USR-Aufruf wird nun den Start des Programmes bewirken. Nun zum Programm selbst. In Bild 17 finden Sie ein Flußdiagramm dazu.

Zunächst konstruieren wir den Teil, der die erste Bildschirmzeile nach $E000 und folgende Speicherstellen schiebt. Das X-Register verwenden wir als Index und laden es mit dez.40 = $27.

Schalten Sie also den SMON ein und starten Sie den Assembler mit:

```
A 02B6
```

Dann geben Sie ein:

```
02B6   LDX   #27
```

Nun packen wir das letzte Zeichen der obersten Bildschirmzeile in den Akku:

```
02B8   LDA   0400,X
```

In das Y-Register legen wir die dazugehörige Farbe aus dem Bildschirmfarbspeicher:

```
02BB   LDY   D800,X
```

Den Akkuinhalt – also die Bildschirminformation – legen wir nach $E000+$27:

```
02BE   STA   E000,X
```

Dasselbe tun wir mit dem Farbcode, der ab $E028+$27 abwärts gespeichert wird. Leider kann man STY nicht X-indiziert-absolut adressieren (siehe Tabelle 5). Deshalb schieben wir zuerst den Y-Registerinhalt in den Akku:

```
02C1   TYA
02C-
2      STA   E028,X
```

Damit ist das letzte Zeichen der Kopfzeile verschoben. Wir zählen das X-Register um 1 herunter:

```
02C-
5      DEX
```

Der X-Index weist nun auf das vorletzte Zeichen, mit dem sich alles ab $02B8 wiederholt. Wenn das X-Register bis 0 heruntergezählt ist, weist es auf das erste Zeichen der Kopfzeile. Die Schleife muß dann noch einmal durchlaufen werden und ein weiteres Herabzählen des X-Registers erzeugt $FF, was zum Setzen der N-Flagge führt. Das ist dann unser Signal, daß die gesamte Kopfzeile übertragen wurde. Die N-Flagge wird durch den BPL-Befehl getestet:

```
02C-
6      BPL   02B8
```

So weit, so gut. Wir hätten natürlich auch das X-Register von 0 an hochzählen können. Zum Beenden der Schleife wäre dann aber ein CPX-Befehl erforderlich gewesen, der jedesmal den X-Registerinhalt mit der Zahl $27 vergleicht.

MERKE: Indexregister in Schleifen abwärts zu zählen, kann Rechenzeit einsparen!

Ab $02CE soll der umgekehrte Vorgang, also das Zurückschieben der vorher gespeicherten Kopfzeile in den Bildschirmspeicher geschehen. Das einfachste wäre es sicherlich, diesen Programmteil mit einem weiteren USR-Kommando zu starten. Das sähe dann so aus:

1.USR-Befehl – schiebt Kopfzeile unter oberes ROM
2.USR-Befehl – holt Kopfzeile zurück in Bildschirmspeicher
3.USR-Befehl – schiebt wieder Kopfzeile unter ROM
4.USR-Befehl – holt sie wieder zurück und so weiter.

Weil aber das Umstellen des USR-Vektors durch POKEs vom Basic aus lästig ist, tun wir das einfach immer am Ende des betreffenden Maschinenprogrammabschnittes. Wir

```
1 REM **********************************     <250>
2 REM *                                *     <229>
3 REM *  TEST FUER DIE 1. VERSION DES  *     <139>
4 REM *        PROGRAMM-PROJEKTES      *     <048>
5 REM * V E R S C H I E B E N  V O N   *     <009>
6 REM *        SPEICHERBEREICHEN       *     <193>
7 REM *                                *     <234>
8 REM * HEIMO PONNATH    HAMBURG 1984  *     <081>
9 REM **********************************     <002>
10 REM                                       <153>
15 REM ++++++ USR-VEKTOR EINSTELLEN ++++     <065>
20 REM                                       <163>
25 POKE 785,182:POKE 786,2                   <239>
30 REM                                       <173>
35 REM ++++++ KOPFZEILE ++++++++++++++++     <013>
40 REM                                       <183>
45 PRINT CHR$(147)CHR$(18)"TEST
   : BILD $0400=1024, FARBE $D800=55296"CHR$(14
   6)                                        <071>
50 PRINT:PRINT:PRINT"DURCH IRGENDEIN USR-KOMMAN
   DO WIRD NUN IM PROGRAMM-MODUS"            <020>
55 PRINT"DER ERSTE TEIL DES VERSCHIEBE-PROGRAMM
   ES AUFGERUFEN"                            <110>
60 PRINT"DIE KOPFZEILE WIRD UNTER DAS OBERE
    ROM{2SPACE}KOPIERT."                     <215>
65 REM                                       <208>
70 REM ++++++ 1. USR-AUFRUF ++++++++++++     <042>
75 REM                                       <218>
80 A=USR(1)                                  <124>
85 PRINT:PRINT"HIER GESCHIEHT DAS DURCH A=USR(1
   ) IN{4SPACE}ZEILE 65"                     <132>
90 PRINT"DABEI IST 1 EIN DUMMY UND MIT A FANGEN
   {2SPACE}WIR AUCH NICHTS WEITER AN."       <063>
95 PRINT"AUF TASTENDRUCK WIRD DER BILDSCHIRM
   {2SPACE}GE-LOESCHT"                       <029>
100 REM                                      <243>
105 REM ++UEBERSCHREIBEN DER KOPFZEILE ++    <046>
110 REM                                      <253>
115 POKE 198,0:WAIT 198,1:PRINT CHR$(147)    <090>
120 REM                                      <007>
125 REM +++ NEUBEGINN DES PROGRAMMES ++++    <173>
130 REM                                      <017>
135 PRINT CHR$(19)"WAS AUCH IMMER JETZT IN DER
    KOPFZEILE{3SPACE}STEHT, ES WIRD BEIM 2.USR"
                      <017>
140 PRINT"VON DEM ZUVOR DURCH DAS ERSTE USR
    GE-{3SPACE}SPEICHERTE UEBERSCHRIEBEN"    <078>
145 PRINT:PRINT"WENN SIE JETZT EINE TASTE DRUEC
   KEN..."                                   <104>
150 POKE 198,0:WAIT 198,1                    <246>
155 REM                                      <042>
160 REM ++++++ 2. USR-AUFRUF +++++++++++     <090>
165 REM                                      <052>
170 A=USR(1):PRINT                           <169>
175 PRINT"IST DIE ALTE KOPFZEILE ZURUECK IN
    DEN{3SPACE}BILDSCHIRMSPEICHER GESCHOBEN."
                      <164>
180 END                                      <052>
```

Bild 18. Test und Demonstration der Verschieberoutine.
Das Programm zeigt das Ein- und Ausschalten einer Kopfzeile auf dem Bildschirm

schreiben also das LSB der Programmfortführung ($CE) nach $311. Das MSB bleibt unverändert $02.

```
02C8  LDA  #CE
02CA  STA  0311
02CD  RTS
```

Mit dem RTS sind wir wieder im Basic-Programm gelandet, welches nun normal weiterverarbeitet wird. Erst ein neues USR-Kommando – im Programm oder im Direktmodus – startet den zweiten Teil unseres Maschinenprogrammes (weil in $0311, – der Einsprungpunkt des USR-Befehls – die Startadresse der auszuführenden Routine steht).

In diesem 2. Teil müssen wir erst einige Befehle geben, die Sie jetzt vielleicht noch nicht verstehen. Das hängt damit zusammen, daß zum Herauslesen des RAM unter dem ROM das ROM ausgeschaltet werden muß (entspricht POKE 1,53):

```
02CE  LDA  01
02D0  PHA
02D1  LDA  #35
02D3  STA  01
```

(Der PHA-Befehl dient hier zur Zwischenspeicherung des Akku-Inhaltes). Das ist hiermit geschehen und wir kommen wieder in bekannte Gefilde mit der Ausleseschleife:

```
02D5  LDX  #27
02D7  LDA  E000,X
02DA  LDY  E028,X
02DD  STA  0400,X
02E0  TYA
02E1  STA  D800,X
02E4  DEX
02E5  BPL  02D7
```

Damit ist die gesamte gespeicherte Kopfzeile wieder zurückgeholt und wir können das ROM wieder einschalten:

```
02E7  PLA
02E8  STA  01
```

Befehls-wort	Adressierung	Byte-zahl	Code Hex	Code Dez	Takt-zyklen	Beeinflussung von Flaggen
LDA	absolut,X	3	BD	189	4	N,Z
	0-page-abs,X	2	B5	181	4	N,Z
	absolut,Y	3	B9	185	4	N,Z
LDX	absolut,Y	3	BE	190	4	N,Z
	0-page-abs,Y	2	B6	182	4	N,Z
LDY	absolut,X	3	BC	188	4	N,Z
	0-page-abs,X	2	B4	180	4	N,Z
STA	absolut,X	3	9D	157	5	/
	absolut,Y	3	99	153	5	/
	0-page-abs,X	2	95	149	4	/
STX	0-page-abs,Y	2	96	150	4	/
STY	0-page-abs,X	2	94	148	4	/
INC	absolut,X	3	FE	254	7	N,Z
	0-page-abs,X	2	F6	246	6	N,Z
DEC	absolut,X	3	DE	222	7	N,Z
	0-page-abs,X	2	D6	214	6	N,Z
ADC	absolut,X	3	7D	125	4	N,V,Z,C
	absolut,Y	3	79	121	4	N,V,Z,C
	0-page-abs,X	2	75	117	4	N,V,Z,C
SBC	absolut,X	3	FD	253	4	N,V,Z,C
	absolut,Y	3	F9	249	4	N,V,Z,C
	0-page-abs,X	2	F5	245	4	N,V,Z,C
CMP	absolut,X	3	DD	221	4	N,Z,C
	absolut,Y	3	D9	217	4	N,Z,C
	0-page-abs,X	2	D5	213	4	N,Z,C
BIT	absolut	3	2C	44	4	N,V,Z
	0-page-abs.	2	24	36	3	N,V,Z
CLV	implizit	1	B8	184	2	V
NOP	implizit	1	EA	234	2	/
TAX	implizit	1	AA	170	2	N,Z
TAY	implizit	1	A8	168	2	N,Z
TXA	implizit	1	8A	138	2	N,Z
TYA	implizit	1	98	152	2	N,Z
JMP	absolut	3	4C	76	3	/
	indirekt	3	6C	108	5	/
JSR	absolut	3	20	32	6	/

Tabelle 8. Zusammenfassung aller wichtigen Daten der neuen Befehle

Falls nun wieder ein USR-Kommando auftaucht, soll die Kopfzeile mit dem 1. Programmteil unter das obere ROM gelegt werden wie am Anfang. Wir müssen deshalb den USR-Vektor auf $02B6 zurückschreiben:

```
02EA  LDA  #B6
02EC  STA  0311
02EF  RTS
```

Das wärs! Wenn nun im Programm oder im Direktmodus wieder ein USR-Befehl auftritt, kann das Ganze von vorne beginnen. In dieser Version wird jedesmal eine neue Kopfzeile hin- und wieder zurückgeschoben. Wenn Sie eine einmal festgelegte Kopfzeile immer wieder benutzen möchten, dann stellen Sie den USR-Vektor einfach nicht mehr zurück: Lassen Sie also die Befehle bei 02EA und 02EC weg. Das Programm endet in dem Fall mit:

```
02EA  RTS
```

Eine wichtige Bemerkung noch: So bequem der Ort auch ist, an dem unser kurzes Programm steht, er hat einen gravierenden Nachteil: Falls Sie mittels einer RESET-Taste oder per Software einen Basic-Kaltstart durchführen, geht unser Programm flöten! Dieser Speicherbereich wird im Reset-Programm nämlich mit lauter Nullen überschrieben. Deswegen speichern Sie es bitte bald ab.

In Bild 18 finden Sie ein kleines Testprogramm für unsere Verschieberoutine, und in Tabelle 8 eine Zusammenfassung aller wichtigen Daten der neuen Befehle.

33. Wir stapeln

In Kapitel 28 haben wir beim JSR-Befehl schon den Stapel etwas kennengelernt. Aber so ganz genau wissen wir's ja noch nicht, was das ist. Deswegen jetzt mal im Detail: Der Stapel, auch Prozessorstack genannt, ist der Speicherbereich von dezimal 256 ($100) bis dezimal 511 ($1FF), der direkt von unserer CPU verwaltet wird. Das ist also die gesamte Page 1. Ähnlich wie bei der String-Verwaltung geschieht auch hier das Füllen von oben nach unten. Das erste Byte, welches in den Stack geschoben wird, kommt also nach $1FF, das nächste nach $1FE und so weiter. Voll ist der Stapel, wenn auch $100 besetzt wurde (siehe Bild 19).

Warum heißt das Ding nun eigentlich Stapel? Das erklärt sich aus dem Zugriffs-Prinzip. Man spricht von einer LIFO-Struktur, von »Last In – First Out«, zu deutsch »zuletzt hinein – zuerst heraus«. Das zuerst hineingebrachte Byte befindet sich am Speicherboden ($1FF), das zuletzt eingebrachte an der Speicherspitze. Stellen Sie sich einen Stapel Akten vor (Bild 20).

Offensichtlich wurde der 4. Aktenordner zuletzt auf den Stapel gesteckt. Er kann zuerst heruntergeholt werden. An die Akte 1 kommen wir erst heran, wenn alle anderen heruntergenommen worden sind. Genauso verhält es sich mit dem Prozessorstack: Um an das unterste Byte des Stapels heranzukommen, müssen erst Byte für Byte die darüberliegenden (nach Bild 19 eigentlich die darunterliegenden) weggeschafft werden.

Mit dem Prinzip des Stapelspeichers werden Sie sich auskennen, wenn Sie schon mal andere Programmiersprachen als Basic ausprobiert haben: In Forth beispielsweise operieren Sie ständig mit Stapeln.

Damit wir – und der Prozessor – den Überblick über den Stack behalten, gibt es dankenswerterweise noch einen Stapelzeiger (stackpointer), der jeweils auf den nächsten freien Platz des Stapels weist. Da gibt's nun aber ein kleines Problem: Der Stapel belegt die komplette Seite 1.

Ein Stapelzeiger, der auf zum Beispiel $01FE zeigen soll, müßte das MSB (also 01) und das LSB (also FE) in zwei Bytes lagern. Der Stapelzeiger ist aber nur 8 Bit groß ... Freundlicherweise sorgt unser Mikroprozessor automatisch für das

neunte Bit. Der Zeiger zählt also immer von $FF an rückwärts bis $00 und weist dabei von $1FF bis $100.

Der Stack hat in unserem Computer drei Aufgaben zu erfüllen:
1) Organisation von Unterprogramm-Adressen
2) Zwischenspeicherung bei Unterbrechungen (Interrupts)
3) vorübergehende Datenspeicherung

Die Rolle des Stapels bei Unterprogramm-Aufrufen haben wir in der letzten Folge schon ausgiebig behandelt. Die sogenannten Interrupts heben wir uns noch für später auf – dazu fehlen uns noch ein paar Kenntnisse. Mit der vorübergehenden Speicherung von Daten befassen wir uns gleich, wenn wir an die Befehle zur Stackbehandlung herangehen.

Zuvor – weil das hier gerade ganz gut paßt – noch ein paar Gedanken zur rekursiven Programmierung. Gemeint ist damit eine Programmstruktur, in der sich ein Unterprogramm selbst aufruft. Auch GOSUB-Befehle in Basic bewirken Einträge der Rücksprungadressen im Stapel. Auf diese Weise ergibt sich für unseren Computer eine begrenzte Verschachtelungstiefe bei Unterprogrammaufrufen. Diese wird bei Rekursion besonders schnell erreicht, und das bewirkt die Ausgabe einer OUT OF MEMORY-Fehlermeldung.

34. Aktives Stapeln mit PHA, PLA, PHP, PLP, TSX und TXS

Mit dem Stapel haben wir 256 Speicherplätze für eine schnelle Zwischenspeicherung aller möglichen Daten zur Verfügung. Weil der 6510 (und natürlich auch der 6502) diesen Speicherbereich wie die Zeropage behandelt, geht das Speichern sehr schnell. Man muß nur immer die spezielle LIFO-Struktur berücksichtigen.

Im Grunde braucht man eigentlich nur zwei Befehle: Etwas auf den Stapel schieben (in der Literatur oft als Push-Befehl bezeichnet) und etwas herunterziehen, das nennt man dann Pull- oder auch Pop-Befehl.

Unser Prozessor kennt insgesamt sechs auf den Stapel wirkende Anweisungen:

PHA Damit schreibt man den Akku-Inhalt in den Stapel (»PusH-Accumulator«). Der Stapelzeiger wird automatisch eine Position heruntergezählt (er rechnet ja von $FF an abwärts!). Der Inhalt des Akku wird dabei nicht verändert. Deswegen bleibt auch das Status-Register (also die ganzen Flaggen: N V B D I Z C) unbeeinflußt.

PLA »PuLl Accumulator«. Das ist der umgekehrte Weg: Das, was zuoberst auf dem Stapel liegt, wird in den Akku geschrieben. Dadurch wird ein Stapelplatz frei, was den Stapelzeiger veranlaßt, um 1 zu wachsen. Weil das, was da in den Akku geladen wird, 0 sein kann oder auch negativ (also mit gesetztem Bit 7), wird unter Umständen auch die N- oder die Z-Flagge verändert.

Weniger mit Datenzwischenspeicherung haben die anderen Befehle zur Stapel-Manipulation zu tun:

PHP Das steht für »PusH Processor status«, also »schiebe das Prozessor-Status-Register auf den Stapel«. Der aktuelle Flaggenstand kann damit aufbewahrt werden. Das Status-Byte ändert seinen Inhalt dabei ebensowenig wir der Akku bei PHA. Auch hier wird der Stapelzeiger freundlicherweise um 1 herabgezählt.

PLP »PuLl Processor status«, »hole den Prozessor-Status vom Stapel« ist der umgekehrte Befehl, der (wie bei PLA in den Akku) das, was zuoberst im Stapel liegt, in das Flaggen-Register schreibt. Da sollte man höllisch aufpassen, was man damit einlädt: Das ist eine feine Gelegenheit für den Computer, abzustürzen. Der Stapelzeiger wird – wie gehabt – um 1 erhöht.

Nicht direkt mit dem Stapel, sondern mit dem Stapelzeiger befassen sich die beiden folgenden Befehle:

TSX »Transfer Stack-pointer into X«, zu deutsch, »schiebe den Stapelzeiger ins X-Register« eröffnet die Möglichkeit, den Stapelzeiger zu lesen. Dabei bleibt er selbst unverändert erhalten. Weil nun im X-Register alle Werte zwischen $FF und 0 auftreten können, werden auch die Flaggen beeinflußt (N- und Z-Flagge).

TXS Den umgekehrten Weg geht »Transfer X into Stackpointer« = »übertrage X-Register-Inhalt in den Stapelzeiger«. Das ist der einzige Befehl, der es erlaubt, den Stapelzeiger mit einem von uns kontrollierten Wert zu laden. Der Inhalt des X-Registers bleibt dabei unverändert, demzufolge interessieren sich auch die Flaggen nicht dafür.

Alle sechs Anweisungen bestehen nur aus einem Byte und sind implizit adressiert. Die Stapelzeiger-Befehle TXS und TSX benötigen zwei Taktzyklen, die Push-Befehle je drei und die Pull-Befehle vier Taktzyklen zur Bearbeitung.

Es ist etwas schwierig, Stapel-Operationen direkt zu verfolgen. Die meisten Assembler – so anscheinend auch der SMON – gebrauchen ebenfalls diesen Speicherbereich. Verlangt man beispielsweise mit dem SMON-Kommando M 0100 01FF eine Darstellung des Stapelinhaltes, dann findet man eine ganze Menge Spuren der Arbeit des Assemblers. Versucht man die zu löschen oder zu überschreiben, zum Beispiel mit dem nachfolgenden kleinen Programm, dann hat der Assembler die Mühe schon wieder zunichte gemacht, wie man durch erneutes M 0100 01FF schnell sehen kann. Dieses kleine Programm soll unterhalb des durch den Stapelzeiger bezeichneten Bereichs 32 Nullen in den Stapel schreiben:

```
8000   LDA   #00
8002   TSX
```

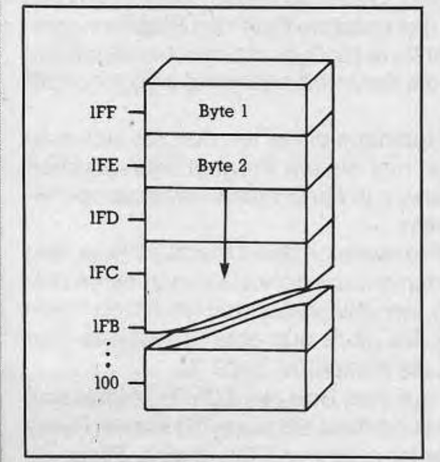


Bild 19. So wird der Stapel gefüllt

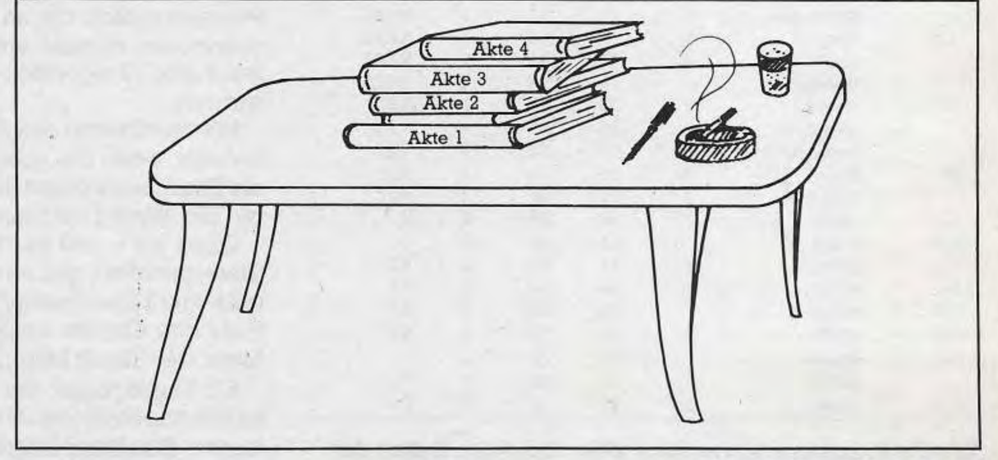


Bild 20. Der Aktenstapel

Der Stapelzeiger wird ins X-Register gerettet.

```
8003   LDY   #20
8005   PHA
```

Wir schieben eine Null auf den Stapel.

```
8006   DEY
8007   BNE   8005
8009   TXS
```

Nach 32 Eintragungen von Nullen stellen wir den alten Stapelzeiger wieder her.

```
800A   BRK
```

Erneutes Kommando M 0100 01FF zeigt keine Nullen. Erst wenn wir anstelle des TXS in Zeile 8009 ein BRK schreiben, den Stapelzeiger also nicht zurückschreiben, erscheinen unsere Nullen. Sieht man genau hin, dann stellt man fest, daß unterhalb des durch den Stapelzeiger bezeichneten Bereichs genau der gleiche Inhalt zu finden ist wie vorher, nur eben mit dem Stapelzeiger verschoben.

Ganz konnte ich dies Rätsel noch nicht lösen, muß ich gestehen, aber für den Gebrauch des Stapels ändert sich dadurch für uns nichts. Worauf muß man achten bei Stapeloperationen? Ganz einfach: Zwischen dem Ablagern eines Wertes auf dem Stapel und dem Zurückholen muß für jeden Push-Befehl ein Pull-Befehl vorhanden sein, für jedes weitere PHA ein PLA, für jedes JSR ein RTS. Nur wenn wir auf diese Symmetrie der Push- und der Pull-Befehle achten (und wie Sie noch aus der vorhergegangenen Ausgabe wissen, sind ja JSR und RTS ebenfalls dazuzurechnen), können wir sicher sein, daß der Stapelzeiger zum Zeitpunkt des Rückholens eines Wertes vom Stapel auch wirklich darauf deutet. Wenn man also nicht ganz genau weiß, wie der verwendete Assembler den Stapel nutzt, sollte man auf Operationen mit den Befehlen TSX und TXS verzichten.

Nun können Sie schon einen Teil der bislang unbekannten Programmsequenz aus der letzten Folge verstehen. Im zweiten Programmteil hatten wir mit

```
02CE   LDA   01
02D0   PHA
```

den Inhalt der Speicherstelle 01 in den Akku geladen und auf den Stapel geschoben. Später – nach einigen weiteren Operationen – wurde dann dieser Speicherinhalt wiederhergestellt durch

```
02E7   PLA
02E8   STA   01
```

Was aber hat es mit dieser Speicherstelle 01 auf sich? Das soll nun als nächstes erklärt werden.

35. Sein oder Nichtsein: Das Rätsel des Prozessorports

Der Commodore 64 hat 64 KByte an RAM zu bieten. Außerdem aber verfügen wir beim normalen Programmieren über weitere 24 KByte, in denen das Betriebssystem, der Basic-Interpreter, Ein- und Ausgabebausteine und der Zeichenspeicher stecken. Wie Sie wissen, umfaßt der Adreßbus aber nur 16 Bit, was uns lediglich 65536 Speicherzellen, also 64 KByte adressieren läßt. Des Rätsels Lösung liegt darin, daß einige Adressenbereiche mehrfach belegt sind. Man kann das vergleichen mit dem Trick des Kastens mit dem doppelten Boden. Welcher Kasteninhalt gerade dem Prozessorzugriff offensteht, wird durch den Prozessorport, das sind die Speicherstellen 00 und 01, gesteuert.

Dr. Helmuth Hauck hat in seiner Serie »Memory Map mit Wandervorschlägen« (64'er, Ausgabe 11 (1984), Seite 135 ff.) die genaue Funktion jedes Bits dieser beiden Speicherstellen erklärt. Wer noch mehr wissen möchte – auch über die Wirkungsweise der beiden Leitungen »Game« und »Exrom« – sollte das nachlesen im »Commodore 64 Programmers Reference Guide« ab Seite 260. Für uns als angehende Assembler-Alchimisten ist die Speicherstelle 1 aber so wichtig, daß wir ganz kurz hier nochmal darauf eingehen.

Die Speichersteuerfunktionen haben die Bits 0 bis 2 der Speicherstelle 1. Je nach Belegung dieser Bits gestaltet sich die 64-KByte-Landschaft unseres Computers wie in Tabelle 9 gezeigt.

Was können wir als Maschinen-Programmierer mit dieser Kenntnis anfangen? Theoretisch stehen uns für unsere Programme damit 64 KByte offen. Praktisch werden wir nur in den seltensten Fällen auf die Ein- und Ausgabe-Bausteine verzichten können. Lassen wir ein reines Maschinenprogramm laufen, ohne jeglichen Rückgriff auf Interpreter oder Betriebssystem, dann haben wir immerhin noch zirka 60 KByte zur freien Verfügung. Benutzen wir Routinen aus diesen beiden ROM-Bausteinen, dann müssen wir sie allerdings – zumindest für den Zeitpunkt des Routineaufrufs – wieder einschalten. Wenn wir – was wohl meistens der Fall sein wird – Kombinationen von Basic- und Assemblersprache verwenden, können wir den gesamten Basic-Speicher bis \$A000 frei halten, können auch den bei allen Beispielprogrammen so beliebten Bereich \$C000 bis \$D000 leer lassen und packen unsere Routinen weitgehend unter die ROMs, die dann jeweils beim Aufruf abgeschaltet werden. So haben wir eine Menge zusätzlichen Speicherplatz ergattert.

Nun können wir auch den letzten Rest des bislang unklaren Programms aus Kapitel 32 verstehen. Nachdem wir den Inhalt der Speicherstelle 1 auf den Stapel gerettet haben (Zeilen \$02CE und \$02D0), schreiben wir \$35 in den Prozessorport:

```
02D1   LDA   #35
02D3   STA   01
```

\$35 ist binär 0011 0101. Die Bits 0 bis 2, auf die es uns in diesem Zusammenhang ankommt, bewirken nun das Ausschalten des Interpreters und des Betriebssystems. Die Ein- und Ausgabe-Bausteine bleiben aktiv. Im weiteren Programmverlauf lesen wir die Speicherinhalte ab \$E000, wobei wir nun den RAM-Inhalt erfassen. Das sollte vielleicht nochmal klargestellt werden: Jedes Hineinschreiben in die mehrfach belegten Speicherbereiche (dabei sind die Ein- und Ausgabe-Bausteine aber ausgenommen) wird automatisch in den RAM-Bereich umgelenkt. Das ist ja auch klar: In ein ROM kann eben nicht geschrieben werden. Deshalb braucht man dabei die ROMs nicht auszuschalten. Jeder Lesevorgang greift aber auf die ROMs zu, weshalb man sie in unserem Fall ausschalten muß. Wie schon oben beim Stapel erklärt, schalten wir durch das Zurückholen des vorher dorthin geretteten alten Inhalts der Speicherstelle 1 in den Prozessorport wieder den Normalzustand ein.

36. Die indirekte Adressierung

Wir werden nun die beiden letzten noch ausstehenden Arten der Adressierung kennenlernen. Beides sind indirekte Adressierungsarten. Mit dem indirekten JMP-Befehl (zum Beispiel

Speicherstelle 1 Bits 2	1	0	\$A000-\$BFFF	\$D000-\$DFFF	\$E000-\$FFFF
1	1	1	Basic	I/O	Kernel
1	1	0	RAM	I/O	Kernel
1	0	1	RAM	I/O	RAM
1	0	0	RAM	RAM	RAM
0	1	1	Basic	Zeichen	Kernel
0	1	0	RAM	Zeichen	Kernel
0	0	1	RAM	Zeichen	RAM
0	0	0	RAM	RAM	RAM

Tabelle 9 zeigt, welche Bausteine bei verschiedener Belegung der Bits 0 bis 2 des Prozessorports (Speicherstelle 1) eingeschaltet sind. (Frei nach Dr. Hauck, 64'er Ausgabe 11/84, Seite 136)

JMP(0300)) sind wir in Kapitel 28 schon vertraut geworden. Wir hatten auch gelernt, daß es sich hierbei um einen absoluten Einzelgänger handelt, der nur für so einen Sprung erlaubt ist. Ebenso haben wir die indizierte Adressierung zu beherrschen gelernt: Das war die Sache mit den Indexregistern X oder Y. Eine Kombination aus beiden (also der indirekten und der indizierten) Adressierungsarten sind die indiziert-indirekte und die indirekt-indizierte Adressierung.

Die indirekt-indizierte Adressierung

Fangen wir mit der sehr häufig benutzten indirekt-indizierten Adressierung an: Man nennt sie auch »indirekt Y« oder »nach-indizierte indirekte« Adressierung. Am besten sehen wir uns mal so einen Befehl an:

```
LDA    (FA),Y
```

Die Klammer erinnert uns an den indirekten JMP-Befehl. Tatsächlich hat sie hier auch dieselbe Funktion: In FA und FB steht ein Zeiger auf eine Adresse. Nehmen wir mal an, die Belegung der Speicher wäre:

```
FA    01
FB    80
```

und im Y-Register stünde eine 5. Der Zeiger FA/FB weist also auf die Speicherstelle 8001. Da haben wir also wieder das Prinzip des toten Briefkastens. Der Computer guckt in den hohlen Baum FA/FB (LSB in FA, MSB in FB) und findet dort die Treffpunktadresse. Nun sind diese toten Briefkästen aber auch den gegnerischen Alchimisten-Agenten bekannt. Es kommt also noch ein Trick dazu: Zur dort aufgefundenen Adresse wird der Inhalt des Y-Registers addiert. In unserem Fall fanden wir also in FA/FB die Adresse 8001, im Y-Register steht eine 5, somit ist die endgültige Adresse 8001+5 = 8006. Unser Beispiel »LDA(FA),Y« bewirkt daher, daß in den Akku der Inhalt der Speicherstelle 8006 geladen wird. Nach-indiziert nennen manche die Adressierung deswegen, weil zunächst dem Zeiger nachgegangen wird, der in unserem Beispiel auf 8001 weist, und erst danach durch Addition des Inhalts des Y-Registers die endgültige Speicherstelle (hier also 8006) berechnet wird.

Als Zeiger (also die Adresse in der Klammer) sind nur Zeropagespeicherstellen verwendbar, als Indexregister darf man hier nur das Y-Register gebrauchen. Von den bisher behandelten Befehlen können ADC, CMP, LDA, SBC und STA mit dieser Adressierungsart verwendet werden. Genaueres finden Sie wieder in der Tabelle mit der Befehlsübersicht (Tabelle 10).

Bevor wir uns dem anderen indirekten Adreß-Modus zuwenden, wollen wir uns überlegen, wozu man die indirekt-indizierte Adressierung verwendet. Wie Sie sich natürlich erinnern können, konnte man mit der normalen indizierten Adressierung, zum Beispiel mit

```
LDA 8000,Y
```

durch Variation des Indexregisters (hier das Y-Register) 256 Speicherstellen erfassen (Y von FF herunter bis 00). Will man mehr als diese 256 berücksichtigen, dann muß eine neue Basis (im Beispiel also anstelle der 8000) gewählt werden. Um das zu illustrieren, sehen wir uns mal den Anfang eines Programms an, welches den gesamten Bildschirminhalt ausliest und nach E000 schreibt:

```
1000   LDY   #00
1002   LDA   0400,Y
1005   STA   E000,Y
1008   LDA   0500,Y
100B   STA   E100,Y
100E   LDA   0600,Y
1011   STA   E200,Y
1014   LDA   0700,Y
1017   STA   E300,Y
101A   DEY
101B   BNE   1002
...
```

Wie Sie sehen, erfordert das durch die Tatsache, daß vier Blöcke zu je 256 Bytes übertragen werden müssen, immerhin schon 28 Bytes Programmtext. Nun soll die indirekt-indizierte Adressierung verwendet werden, um dieselbe Aufgabe zu lösen. Wir legen zunächst zwei Zeiger auf der Zeropage fest:

FA/FB sollen die Bildschirmadresse enthalten
FC/FD die Zieladresse ab E000.

```
1000   LDA   #00
1002   STA   FA
1004   STA   FC
```

Das waren die LSBs der Zeiger, es folgen die MSBs:

```
1006   LDA   #04
1008   STA   FB
100A   LDA   #E0
100C   STA   FD
```

Damit sind die Zeiger festgelegt. Es sind vier Blöcke zu je 256 Bytes zu übertragen. Diese Blockanzahl legen wir ins X-Register als Zähler:

```
100E   LDX   #04
```

Dann laden wir ins Y-Register ebenfalls einen Zähler (den Index):

```
1010   LDY   #00
```

Jetzt kann die eigentliche Übertragungsschleife starten:

```
1012   LDA   (FA),Y
1014   STA   (FC),Y
1016   DEY
1017   BNE   1012
```

Wenn das Y-Register wieder bei 0 angekommen ist (von der ersten 0 nach einem Unterlauf über FF, FE und so weiter bis 0), ist der erste Block übertragen. Wir erhöhen nun das MSB beider Zeiger um 1:

```
1019   INC   FB
101B   INC   FD
```

Außerdem zählen wir den Blockzähler um 1 herunter:

```
101D   DEX
101E   BNE   1012
...
```

Wenn das Programm auf diese Weise auch drei Byte mehr Speicherplatz braucht, ist doch leicht der Vorteil zu sehen: Müssen wir nämlich (statt nur vier) mehr Blöcke übertragen (bis zu 255), dann verändert sich unser zweites Programm um keinen Deut (außer dem Zähler im X-Register, der nun mit der jeweils anderen Block-Anzahl geladen wird), während die

Befehlswort	Adressierung	Bytezahl	Code Hex	Code Dez	Taktzyklen	Beeinflussung von Flaggen
LDA	indirekt X	2	A1	161	6	N,Z
	indirekt Y	2	B1	177	5*	N,Z
STA	indirekt X	2	81	129	6	—
	indirekt Y	2	91	145	6	—
ADC	indirekt X	2	61	97	6	N,V,Z,C
	indirket Y	2	71	113	5*	N,V,Z,C
SBC	indirekt X	2	E1	225	6	N,V,Z,C
	indirekt Y	2	F1	241	5*	N,V,Z,C
CMP	indirekt X	2	C1	193	6	N,Z,C
	indirekt Y	2	D1	209	5*	N,Z,C
PHA	implizit	1	48	72	3	—
PLA	implizit	1	68	104	4	N,Z
PHP	implizit	1	08	8	3	—
PLP	implizit	1	28	40	4	alle
TSX	implizit	1	BA	186	2	N,Z
TXS	implizit	1	9A	154	2	—

* Wenn bei der Befehlsausführung eine Page-Grenze überschritten wird, muß noch ein Taktzyklus dazugerechnet werden.

Tabelle 10. Übersicht der in dieser Folge vorgestellten Befehle

erste Programmtechnik für jeden weiteren Block um sechs Bytes erweitert werden muß.

Es gibt noch eine ganze Reihe von Anwendungsmöglichkeiten, die die indirekt-indizierte Adressierung so attraktiv machen. Für Geschwindigkeitsfanatiker (ich selbst bin bei Grafik-Fragen auch einer!) muß aber gesagt werden, daß dem Speicherplatzvorteil ein Geschwindigkeitsnachteil gegenübersteht. Jeder indirekt-indiziert adressierte Befehl braucht einen Taktzyklus länger als der vergleichbare absolut-indizierte Befehl. Zu diesen Feinheiten werden wir aber später noch kommen.

Die indiziert-indirekte Adressierung

Wenden wir uns nun der letzten noch fehlenden Adressierungsart zu, der indiziert-indirekten. Man nennt sie auch »vorindizierte indirekte« oder »indirekt X« Adressierung. Sehen wir auch hier zunächst ein Beispiel an:

```
STA   (FA,X)
```

Auch hier drückt die Klammer wieder aus, daß der Klammerinhalt ein Zeiger ist. Das ist jetzt aber nicht das Bytepaar FA/FB, sondern zur angegebenen Adresse FA soll noch der Inhalt des X-Registers addiert werden. Nehmen wir mal an, dort stünde eine 2, dann wird der Zeiger FC/FD mit diesem Befehl angesprochen, denn FA+2=FC und entsprechend FB+2=FD. Wenn in den Speicherstellen FA bis FF folgender Inhalt zu finden ist:

```
OOFA   00
OOFB   04     FA/FB = 0400
OOFC   00
OOFD   E0     FC/FD = E000
OOFE   10
OOFF   80     FE/FF = 8010
```

dann könnte das eine ganze Tabelle von Zeigern sein, die jeweils durch den X-Registerinhalt angesprochen werden. Der Akkuinhalt wird in unserem Beispiel nach 0400 geschrieben, wenn im X-Register 0 steht, nach E000, wenn das X-Register eine 1 enthält und nach 8010, wenn stattdessen eine 2 im X-Register zu finden ist.

Sie werden sich vielleicht auch bei diesem Beispiel gefragt haben, was passiert, wenn im X-Register unseres Beispiels eine 6 steht. Nun, unser 8-Bit-Prozessor läuft über, und wir finden einen Zeiger 00/01.

Rein theoretisch ist diese Adressierungsweise ganz interessant. Aber auf der Zeropage ist's reichlich eng, und nur selten kommt man daher in die Lage, dort eine Zeigertabelle einzurichten, die man mittels des X-Registerinhalts und der indiziert-indirekten Adressierung abgreifen kann. Die Bedeutung dieser Adressierungsart ist also nur recht gering. Außerdem erfordert sie sechs Taktzyklen zur Bearbeitung und ist somit auch noch recht langsam. Von den bisher bekannten Befehlen sind die folgenden damit verwendbar: ADC, CMP, LDA und STA.

Bevor wir die Adressierung zu den Akten legen, sei noch erwähnt, daß manche Lehrbücher noch eine weitere Art, die Akkumulator-Adressierung, unterscheiden. Betroffen sind davon vier 1-Byte-Befehle, die wir noch kennenlernen werden und die man ebensogut als implizit adressiert ansehen kann.

37. Die ersten Kernel-Routinen

Sicher werden Sie alle schon von der Kernel-Routine FFD2 gehört haben und sie vielleicht auch schon verwenden. Wenn nicht, um so besser, denn dann sind Sie noch nicht vom einseitigen Gebrauch dieses Instruments verdorben. Die meisten Kernel-Adressen sind nämlich sehr vielseitig verwendbar, je nach den Vorgaben. Das ist wie mit einem Haushaltsgerät, das immer nur zum Rühren von Kuchenteig eingesetzt wird. Dabei kann man damit auch noch Saft machen, Gurken schnitzeln, Getränke mixen ... Genauso wie man in diesem etwas schiefen Vergleich die Gebrauchsanleitung kennen sollte, um die ganzen anderen Funktionen ausnutzen zu können, muß man hier noch einige Dinge über die Kernel-Aufrufe beherzigen.

Für jede Verwendung der Kernel-Sprungtabelle sollte man sich angewöhnen, dies in drei Schritten zu tun:
1) die nötigen Vorbereitungen treffen
2) Routineaufruf
3) Fehlerabfrage und -behandlung

Fangen wir mit dem Punkt »Vorbereitungen« an. Einige Routinen brauchen Informationen, die ihnen erst durch andere Kernel-Routinen beschafft werden. Ruft man diese anderen Routinen vorher nicht auf, dann funktioniert auch der erwünschte Aufruf nicht richtig. Wenn die Routine einen bestimmten Wert im Y-Register erwartet, dann muß der dort auch stehen. Wenn nicht, dann geht das Programm in die Hose. Bei jeder Kernel-Routine, die hier beschrieben wird, gebe ich alle nötigen Vorbereitungen an.

Der Routinenaufruf sollte immer mittels JSR erfolgen. Alle auf diese Weise aus der Kernel-Sprungtabelle abzurufenden Programme enden nämlich mit einem RTS. Damit keine wichtigen Werte aus dem Aufrufprogramm überschrieben werden, man sie also vor dem Aufruf der Kernel-Routine irgendwohin retten kann, gebe ich auch noch an, welche Register durch die Routine verändert werden und wieviel Stapelspeicherplatz bereitgehalten werden muß.

Die Routinen sind so konstruiert, daß beim Auftreten eines Fehlers nach der Rückkehr das Carry-Bit gesetzt ist. Durch Untersuchen des Carry können so Fehler rechtzeitig erkannt und behandelt werden. Im Akku findet man in dem Fall dann eine Fehlernummer. Die Ausgabe der Fehlermeldung erfolgt also nicht – wie im Basic – in Klarschrift. In Tabelle 11 sind die Fehlernummern und ihre Bedeutung aufgelistet.

Welche Fehlernummern eine Routine ausgeben kann, wird ebenfalls von mir bei jeder Routinen-Besprechung angegeben.

Nun aber zur ersten Routine FFD2, die wie einen Rattenschwanz eine Reihe weiterer nach sich zieht:

Name	CHROUT
Zweck	Ausgabe eines Zeichens
Adresse	$FFD2 dez. 65490
Vorbereitungen	(CHKOUT,OPEN) Zeichen im Akku
Fehler	0
Stapelbedarf	8
Register	Akku

Falls Sie diese Routine schon einmal benutzt haben, dann geschah es vermutlich ohne die Vorbereitungen CHKOUT und OPEN. Freundlicherweise hat unser Computer einige Voreinstellungen schon für uns getroffen. Denn normaler-

Nummer	Text	Bedeutung
0	BREAK	Während des Programms wurde die RUN/STOP-Taste gedrückt
1	TOO MANY FILES	Man kann maximal 10 offene Files einrichten
2	FILE OPEN	Ein bereits geöffnetes File wird nochmals geöffnet
3	FILE NOT OPEN	Auf ein noch nicht geöffnetes File sollte zugegriffen werden
4	FILE NOT FOUND	Das geforderte File ist nicht verfügbar
5	DEVICE NOT PRESENT	Das angesprochene Gerät zeigt keine Reaktion
6	NOT INPUT FILE	Aus einem Schreibfile kann nicht gelesen werden
7	NOT OUTPUT FILE	In ein Lesefile kann nicht geschrieben werden
8	MISSING FILE NAME	Bei Operationen, die einen Filenamen erfordern, fehlt dieser
9	ILLEGAL DEVICE NUMBER	Das versuchte Kommando ist beim angesprochenen Gerät nicht möglich

Tabelle 11. Fehlernummern und ihre Bedeutung. Die Nummern findet man bei gesetztem Carry im Akku.

weise sendet CHROUT ein Zeichen über einen schon geöffneten Ausgabekanal, und der ist zum Bildschirm geschaltet. Ein kleines Beispielprogramm soll das illustrieren. Zunächst laden Sie bitte den SMON ein und starten Sie ihn. Nun soll eine Texttabelle angelegt werden. Das funktioniert beim SMON am bequemsten über das K-Kommando. Geben Sie ein K 6000. Der SMON antwortet mit:

'6000 ....

Wenn Sie nun die RUN/STOP-Taste drücken, können Sie mit dem Cursor in diese Punktzeile fahren und einen Text schreiben:

'6000 HALLO ASSEMBLER-ALCHIMIST ....

Sinnvoll – vor allem für die weitere Verwendung dieses Textes – ist es, ein (RETURN), also dezimal 13 oder $0D anzuschließen. Dazu gibt es natürlich den Weg über den Assemblerbefehl. Zur Übung wollen wir aber das M-Kommando verwenden. Geben Sie ein (zuerst die »RETURN«-Taste betätigen) M6018, dann wieder RUN/STOP, und fahren Sie mit dem Cursor auf Speicherstelle 601A (falls Sie in 6019 kein Leerzeichen $20 stehen haben, dann fügen Sie's jetzt noch ein). Geben Sie nun anstelle des dort stehenden Bytes 0D ein, und drücken Sie die RETURN-Taste. Der Monitor sollte jetzt zeigen:

```
:6018 54 20 0D
```

etc.

Unser Text soll mit einem BRK enden. Deshalb gehen wir jetzt in den Assembler-Modus mit dem SMON-Kommando A 601B und schreiben:

```
601B  BRK
```

Nun folgt das eigentliche Programmchen, das Byte für Byte bis zur Null (BRK) den Text aus der gerade erstellten Texttabelle liest und mittels FFD2 auf den Bildschirm bringt:

```
601C  LDY  #00
601E  LDA  6000,Y
6021  BEQ  602C
```

Das Y-Register wird als Index initialisiert, dann die Texttabelle in den Akku geladen. Wenn das Programm dabei auf die Null stößt, verzweigt es zum Ende. Jetzt folgt die Routine zur Bildschirmausgabe:

```
6023  JSR  FFD2
6026  BCS  602D
```

Falls bei der Kernal-Routine etwas schiefgelaufen ist, wird das Carry-Bit gesetzt, was wir überprüfen und zu einem BRK-Kommando verzweigen (das ist natürlich nur sinnvoll, solange ein Monitor oder Assembler wie der SMON aktiv ist). Nun erhöhen wir das Index-Register und das ganze beginnt von vorne:

```
6028  INY
6029  JMP  601E
602C  RTS
602D  BRK
```

Wenn wir nun aus dem SMON mit F und anschließendem X aussteigen und ein kleines Basic-Aufrufprogramm machen (Bei OUT OF MEMORY ERROR bitte NEW eingeben):

```
10 PRINTCHR$(147)
20 SYS 24604 :REM = $601C
30 END
```

dann können wir uns die Wirkung unseres Programms ansehen: Nach RUN wird der Bildschirm gelöscht und unser Text ausgedruckt.

FFD2 nimmt uns also eine Menge Arbeit ab: Automatisch legt diese Routine in den Bildschirmspeicher den Bildschirmcode (sie rechnet also auch gleich ASCII, das wir ja eingegeben haben, in den POKE-Code um) und in die dazugehörige Bildschirmfarbspeicherstelle den aktuellen Farbcode. Sie setzt außerdem noch den Cursor weiter.

Mit FFD2 kann man aber noch viel mehr machen! Schließlich ist ja der Bildschirm (Gerätenummer 3) nicht der einzige mögliche Empfänger. Wir wollen als nächstes mal eine Ausgabe mittels FFD2 auf den Drucker erzielen. Hier sind die Vorbereitungen allerdings nötig. Zunächst mal müssen wir uns noch zwei weitere Kernal-Routinen ansehen, nämlich CHKOUT und OPEN.

Name	CHKOUT
Zweck	Kanal zum Ausgang definieren
Adresse	$FFC9 dez. 65481
Vorbereitungen	OPEN log. Filenummer ins X-Register
Fehler	0,3,5,7
Stapelbedarf	4
Register	Akku, X-Register

Mit dieser Routine kann jedes File, der zuvor durch OPEN spezifiziert worden ist, zum Ausgabefile erklärt werden. Natürlich muß dann das derart angesprochene Gerät auch ein Ausgabegerät sein. Andernfalls ergibt sich ein Fehler. Bevor man Daten über einen Kanal senden will, muß CHKOUT durchgeführt werden. Wenn die mittels OPEN übergebene Geräteadresse größer als 3 ist, sendet diese Routine automatisch auch ein LISTEN-Kommando an das Ausgabegerät. LISTEN setzt dann zum Beispiel den Drucker in Empfangsbereitschaft. Die Durchführung von CHKOUT ist einfach (vorausgesetzt, man hat vorher OPEN aufgerufen): In das X-Register wird die logische Filenummer geschrieben und dann per JSR FFC9 (CHKOUT) angesteuert.

Nun zur anderen Vorbereitung von FFD2, zu OPEN:

Name	OPEN
Zweck	Öffnen eines logischen Files
Adresse	$FFC0 dez. 65472
Vorbereitungen	SETLFS,SETNAM
Fehler	1,2,4,5,6
Register	Akku, X- und Y-Register

Die Routine OPEN an sich anzusprechen ist relativ einfach. Es genügt ein JSR FFC0. Zuvor allerdings – der Rattenschwanz wird länger – muß mit SETNAM der Filename und mit SETLFS die logische Filenummer, die Geräteadresse und eventuell eine Sekundäradresse festgelegt sein. Erst danach kann das File geöffnet werden durch OPEN. Also sehen wir uns noch SETLFS und SETNAM an:

Name	SETLFS
Zweck	Spezifikationen eines logischen Files
Adresse	$FFBA dez. 65466
Vorbereitungen	logische Filenummer in Akku Gerätenummer ins X-Register Sekundäradresse ins Y-Register
Fehler	keine
Stapelbedarf	2
Register	keine

SETLFS legt für die anderen Kernal-Routinen logische Filenummer, Gerätenummer und Sekundäradresse fest. Die logische Filenummer ist dabei eine Schlüsselzahl, die in eine durch OPEN angelegte File-Tabelle weist. Die Gerätenummer kann zwischen 0 und 31 liegen, dabei sind folgende Zuordnungen vorgesehen:

0 Tastatur 2 RS232C-Kanal
1 Datasette 3 Bildschirm

Gerätenummern ab 4 beziehen sich automatisch auf Geräte am seriellen Bus. Dabei gilt im allgemeinen:

4 Drucker
8 Diskettenstation

Die Sekundäradresse ist eine Kommandonummer, die für das jeweils angesprochene Gerät spezifisch ist, zum Beispiel 10 bewirkt beim Drucker Commodore 1526, daß das Gerät in die Grundstellung geht (siehe jeweiliges Handbuch). Will man keine Sekundäradresse verwenden, dann muß FF ins Y-Register geschrieben werden. Der Aufruf von SETLFS geschieht also in folgender Weise: In den Akku lädt man die gewünschte logische Filenummer, ins X-Register die Geräteadresse und ins Y-Register FF oder aber die Sekundäradresse. Danach erfolgt der Sprung mit JSR FFBA.

Schließlich noch zu SETNAM:

Name	SETNAM
Zweck	Filenamen festlegen
Adresse	FFBD dez. 65469
Vorbereitungen	Namenslänge in den Akku LSB des Namenstextes in X-Register MSB des Namenstextes in Y-Register
Fehler	keine
Stapelbedarf	2
Register	Akku, X- und Y-Register

Vor der Eröffnung eines Files mittels OPEN muß diese Routine den Filenamen festlegen. Dazu schreibt man in den Akku die Länge des Namens und in die Register X, Y die Startadresse (LSB ins X-Register, MSB ins Y-Register) der Namenstext-Tabelle. Der Ort dieser Tabelle ist frei wählbar. Wird kein Filename gewünscht, dann gibt man dem Akku die Länge 0 an. X- und Y-Register sind in dem Fall ohne Bedeutung.

Damit – sollte man meinen – hätten wir nun alle Bedingungen erfüllt, FFD2 zur Ausgabe auf den Drucker zu bewegen. Leider ist das noch nicht der Fall: FFD2 schließt nämlich das File und den Ausgabekanal nicht. Das kann – wenn man's nicht beachtet – zu Fehlern oder zur weiteren Ansprache des Druckers führen, auch wenn die gar nicht mehr erwünscht ist. Deswegen sollten noch zwei Kernel-Routinen angehängt werden, von denen die eine (CLRCHN) alle Ein- und Ausgabekanäle wieder in den Ausgangszustand zurückführt, und die andere (CLOSE) das File ordnungsgemäß schließt:

Name	CLRCHN
Zweck	Ein- und Ausgabekanäle in Ausgangsstellung bringen
Adresse	$FFCC dez. 65484
Vorbereitung	keine
Fehler	keine
Stapelbedarf	9
Register	Akku, X-Register

Der Aufruf von CLRCHN erfolgt einfach durch JSR FFCC. Die Wirkung ist enorm: Mit einem Schlag werden alle Kanäle freigeräumt. Eingangskanälen wird ein UNTALK (dem Gerät wird gesagt: Halt den Mund), Ausgangskanälen ein UNLISTEN (das bedeutet soviel wie: Hör nicht mehr zu) übermittelt. Der Ausgangszustand stellt sich wieder her: Tastatur als Eingabe-Bildschirm als Ausgabegerät.

Die endgültig letzte Routine für diesmal ist CLOSE:

Name	CLOSE
Zweck	Schließen logischer Files
Adresse	$FFC3 dez. 65475
Vorbereitungen	logische Filenummer in Akku
Fehler	0
Stapelbedarf	2
Register	Akku, X- und Y-Register

Bild 21. Die Abfolge der Routineaufrufe ▶

SETNAM → SETLFS → OPEN → CHKOUT → CHROUT → CLRCHN → CLOSE

Wenn für ein File alle Ein- und Ausgabeoperationen beendet sind, kann es – nach Einschreiben der Filenummer in den Akku – mittels CLOSE ordnungsgemäß geschlossen werden. Der Eintrag in der Filetabelle wird auf diese Weise gelöscht.

So, jetzt sind wir soweit, daß wir die Textausgabe auf dem Drucker programmieren können. Bild 21 faßt die einzelnen Schritte nochmal zusammen.

Und hier das Programm dazu. Wir verwenden die im anderen Beispiel schon aufgebaute Texttabelle weiter. Zunächst also SETNAM:

```
601C  LDA  #00
601E  JSR  FFBD
6021  BCS  6053
```

Wenn ein Fehler aufgetreten ist, findet man ein gesetztes Carry-Bit. In dem Fall wird verzweigt zu einem BRK-Kommando (was die Anwesenheit eines Monitors erforderlich macht, solange man sich noch nicht sicher ist, ob Fehlermeldungen auftauchen). Die Null im Akku besagt, daß kein Filename gewünscht ist. Dann kommt SETLFS:

```
6023  LDA  #04
6025  LDX  #04
6027  LDY  #FF
6029  JSR  FFBA
602C  BCS  6053
```

Es wurde ein File festgelegt mit der logischen Filenummer 4, der Geräteadresse 4 und ohne Sekundäradresse. Jetzt geben wir das OPEN-Kommando:

```
602E  JSR  FFC0
6031  BCS  6053
```

Der Ausgabekanal wird definiert mit CHKOUT:

```
6033  LDX  #04
6035  JSR  FFC9
6038  BCS  6053
```

Damit sind alle Vorbereitungen erledigt und die Zeichenausgabe kann wie im ersten Programm durchgeführt werden mit CHROUT:

```
603A  LDY  #00
603C  LDA  6000,Y
603F  BEQ  604A
6041  JSR  FFD2
6044  BCS  6053
6046  INY
6047  JMP  603C
```

Alle Zeichen sind nun ausgedruckt. Wir rufen CLRCHN auf:

```
604A  JSR  FFCC
```

Als letzte Routine folgt nun noch CLOSE:

```
604D  LDA  #04
604F  JSR  FFC3
6052  RTS
```

Damit wurde das File Nummer 4 geschlossen. Anschließend erfolgte der Rücksprung aus dem Programm. Für die

Fehlerbehandlung habe ich nur einen BRK vorgesehen, der sofortigen Registerüberblick erlaubt, wenn zum Beispiel der SMON im Speicher enthalten ist.

6053 BRK

Ohne Monitor im Speicher kann der Computer allerdings abstürzen oder im besten Fall einen Basic-Warmstart durchführen. Wenn Sie sowas also für Ihre Zwecke programmieren möchten, sollten Sie einen anderen Weg suchen, die Fehler aufzufangen. Man hat ja nicht immer einen Monitor eingeladen.

Mit diesen sieben Kernel-Routinen beenden wir dieses Kapitel. In der 64'er, Ausgabe 12/84 haben B. Schneider und K. Schramm in ihrer Serie »In die Geheimnisse der Floppy eingetaucht« gezeigt, wie man mittels der besprochenen Routinen, und einiger anderer, auch die Diskettenstation ansprechen oder sogar Floppy und Drucker zum »Spooling« veranlassen kann. Das habe ich zwar schon öfter gesagt, muß es aber trotzdem immer wieder tun: Durch das Nachvollziehen fremder Programme kann man sehr viel lernen.

38. Der C 64 und Fließkommazahlen

Inzwischen wissen Sie ja, daß alle Daten im Computer im Binärformat enthalten sind. Wie man eine normale, ganze Zahl zur binären umrechnet, wurde schon im Grafik-Kurs (64'er, Ausgaben 4 und 5 von 1984) gezeigt. Da aber viele Leser dieses Assemblerkurses die genannten Ausgaben nicht besitzen, soll doch nochmal vorgestellt werden, welcher Rechenweg der einfachste ist. Als Beispiel nehmen wir die Zahl 1985. Man teilt diese Zahl so lange durch 2, bis das Ergebnis 0 wird. Jedesmal notiert man sich den Rest, der entweder 0 oder 1 sein kann:

```
1985 : 2=  992 Rest 1
 992 : 2=  496 Rest 0
 496 : 2=  248 Rest 0
 248 : 2=  124 Rest 0
 124 : 2=   62 Rest 0
  62 : 2=   31 Rest 0
  31 : 2=   15 Rest 1
  15 : 2=    7 Rest 1
   7 : 2=    3 Rest 1
   3 : 2=    1 Rest 1
   1 : 2=    0 Rest 1
```

Auch wenn Sie es noch nicht erkennen: Da steht schon das binäre Ergebnis. Von unten nach oben gelesen, ist das nämlich der Rest:

111 1100 0001

Nun reden wir ja von Fließkommazahlen. Also verändern wir unser Beispiel noch etwas. Jetzt soll uns die Zahl 1985,125 interessieren. In der Ausgabe 6/84 haben Sie gelernt, daß man das Komma verschieben kann, um daraus beispielsweise $1,985125 \times 10^3$ zu machen. Wir wollen uns das Verschieben des Kommas aber für etwas später aufheben und zunächst einmal außer dem schon umgewandelten Vorkammateil nun auch den Nachkommateil, also die 0,125, ins Binärformat übertragen.

Genauso, wie wir vorhin eine Kettendivision durch 2 verwendet haben, gebrauchen wir nun eine Kettenmultiplikation mit 2. Der gesamte Nachkommateil wird dabei verdoppelt. Entweder ergibt sich dabei eine Vorkommastelle (das ist dann immer eine 1) oder das Ergebnis bleibt kleiner als 1. Wenn sich bei einem solchen Rechenschritt keine Vorkommastelle ergibt, schreibt man an die entsprechende Nachkommastelle der Binärzahl eine 0, andernfalls eine 1. Es wird solange verdoppelt, bis keine Nachkommastellen mehr zur Verfügung stehen. Das klingt ziemlich umständlich. Am besten sehen Sie sich das jetzt mal an unserem Beispiel an:

0,125 x 2 = 0,250   1. Nachkommastelle:0

Beim ersten Verdoppeln hat sich keine neue Vorkommastelle ergeben, deshalb ist die erste Nachkommastelle der Binärzahl eine Null.

0,25 x 2 = 0,5   2. Nachkommastelle:0

Auch beim zweiten Verdoppeln ermitteln wir keine neue Vorkommastelle, wodurch sich wieder eine Null als Nachkommastelle ergibt.

0,5 x 2 = 1,0   3. Nachkommastelle:1

Hier hat sich nun eine Vorkommastelle beim Verdoppeln gebildet: Daher taucht als 3. Nachkommastelle unserer Binärzahl eine 1 auf. Gleichzeitig war das die letzte Nachkommastelle, denn unsere Ausgangszahl weist nach dem Komma nun nur noch eine Null auf.

Zur Übung wollen wir noch eine andere Zahl mit Nachkommastellen ins Binärformat überführen, nämlich 0,1.

```
0,1x2 = 0,2    1. Nachkommastelle:0
0,2x2 = 0,4    2. Nachkommastelle:0
0,4x2 = 0,8    3. Nachkommastelle:0
0,8x2 = 1,6    4. Nachkommastelle:1
```

Jetzt läßt man – das habe ich beim ersten Beispiel noch nicht erwähnt – diese neue Vorkommastelle einfach weg und rechnet wieder mit den Nachkommastellen weiter:

```
0,6x2 = 1,2    5. Nachkommastelle:1
0,2x2 = 0,4    6. Nachkommastelle:0
0,4x2 = 0,8    7. Nachkommastelle:0
0,8x2 = 1,6    8. Nachkommastelle:1
0,6x2 = 1,2    9. Nachkommastelle:1
```

Das kommt Ihnen sicherlich von der 5. Verdoppelung her bekannt vor. Es zeigt sich, daß diese Rechnung nie aufgeht, weil sich eine periodische Zahl ergibt:

0,000 1100 1100 1100...

Das kann Ihnen öfters bei der Zahlenumwandlung passieren, daß ein endlicher Dezimalbruch in einen unendlichen periodischen Binärbruch übergeht. Kehren wir zurück zu unserem ersten Beispiel, 1985,125. Die ganze Umwandlung (Vorkomma- und Nachkommaanteil) führte zu:

111 1100 0001,001

Der dritte Schritt der Verwandlung von der Dezimalzahl zum Binärformat (nach 1.=Vorkommaanteil umwandeln, 2.=Nachkommaanteil umwandeln) ist das sogenannte Normalisieren. Das ist einfach das Verschieben des Kommas nach links (wie in unserem Beispiel) oder rechts, solange, bis vor dem Komma nur noch Nullen stehen und direkt hinter ihm eine 1. In Kapitel 29 haben wir gelernt, daß für jede Stelle, die das Komma nach links wandert, der Exponent um 1 höher wird. Unser Exponent ist im Moment noch Null (2^0 ist ja 1). Um also nach der Regel zu normalisieren, wird das Komma um 11 Stellen nach links verschoben. Der Exponent ist dann 11(dez) und unsere Zahl erscheint im neuen Gewand:

0.1111 1000 0010 01 E +1011

E +1011 heißt dabei Exponent, und wird im Binärformat dargestellt (1011 (bin.) ≙ 11 (dez.)). Soweit, sogut. Alles bisher unternommene hat Allgemeingültigkeit. Von nun an aber müssen wir uns spezialisieren auf den Commodore 64 (im VC 20 und einigen anderen Computern ist es aber auch so). Der Exponent kann ja – je nach dem, ob das Komma nach links oder nach rechts zum Normalisieren verschoben wurde – positiv sein (wie bei unserem Beispiel) aber auch negativ. Im Commodore 64 wird zum Exponenten die Zahl 128 addiert. Das ist dann Schritt 4, der im Beispiel zu 139 führt, womit wir schon das Exponentenbyte fertig haben:

Exponent: dez.139 bin.1000 1011 hex.8B

Hätten wir einen negativen Exponenten erhalten, zum Beispiel 20, dann stünde im Exponentenbyte nun dez.108, beziehungsweise dasselbe im Binärformat.

Der Rest unserer Zahl, also die Mantisse, wird nun Schritt 5 unterzogen. Zunächst läßt man das Komma weg. Die Binär-

zahl wird dann auf 4 Byte linksbündig aufgeteilt. In unserem Beispiel erhalten wir so:

1111 1000	0010 0100	0000 0000	0000 0000
Byte 1	Byte 2	Byte 3	Byte 4

Wie Sie sehen, werden die unbenutzten Bits mit Nullen aufgefüllt. Was nun noch nicht berücksichtigt wurde, ist das Vorzeichen der Mantisse. Es ist im Beispiel noch nicht zu erkennen, ob wir +1985,125 oder -1985,125 vorliegen haben. Das gehen wir nun im letzten Schritt (Nummer 6) an. Im Commodore 64 gibt es zwei Möglichkeiten der Speicherung von Fließkommazahlen. Für Schritt 6 muß man sich entscheiden, wo man die Zahl haben will.

In Kapitel 30 ist schon mal der FAC erwähnt worden, der Fließkomma-Akkumulator 1, welcher die Speicherstellen dez. 97 bis 102 ($61 bis $66) belegt. Ein zweiter Fließkomma-Akkumulator, AFAC oder ARG genannt, belegt die Plätze dez. 105 bis 110 ($69 bis $6E). Diese Akkumulatoren haben für die Fließkommarechnungen eine ähnliche Bedeutung wie der Akku für die 1-Byte-Rechnungen. Dort werden fast alle Ergebnisse abgelegt oder Zahlen abgerufen. Wir sehen, daß wir darin 6 Byte zur Verfügung haben. In Byte 97 liegt der Exponent in der von uns ermittelten Form. Byte 98 bis 101 sind die vier Mantissenbytes. Was ist in Byte 102? Das Vorzeichen! Bit 7 dieses Bytes ist 0, wenn eine positive, und 1 wenn eine negative Zahl vorliegt. Das galt für den FAC, wie Sie aus den Speicherstellen schon gesehen haben. Für den ARG ist das aber ganz genauso. Sehen wir uns nun in Bild 22 unsere Beispielzahl im FAC und im ARG nochmal an.

Im Bild ist auch angedeutet, daß die restlichen 7 Bit (Bits 0 bis 6) des Vorzeichenbytes keine Rolle spielen. Sie werden später direkt in diese Akkumulatoren hineinsehen und allerlei Bit-Müll darin finden. Lediglich Bit 7 ist für uns von Bedeutung.

Eigentlich ist das ja eine ganz schöne Verschwendung, von einem Byte wie diesem Vorzeichenbyte lediglich ein einziges Bit zu nutzen. Wenn eine beliebige Fließkommazahl irgendwo im Computer abgespeichert wird, dann gilt ein anderes Format, das MFLPT-Format (von Memory-FLoating PoinT). Man speichert hier nur in 5 Byte. Das Vorzeichenbyte fällt weg. Wie aber merkt sich der Computer das Vorzeichen? Das ist ganz schlau eingefädelt: Es gibt nämlich in den 5 Byte (1 Exponentenbyte + 4 Mantissenbyte) ein überflüssiges Bit. Sie werden sich sicher erstaunt fragen, wo?

Erinnern Sie sich doch bitte zurück an den Schritt 3, das Normalisieren. Dort wurde so verfahren, daß rechts vom Komma eine 1 steht. Wenn da aber immer und ganz grundsätzlich diese 1 steht, dann muß man sie sich eigentlich gar nicht mehr besonders merken. Man kann – vorausgesetzt, man berücksichtigt diese 1 im Bit 7 des ersten Mantissen-Bytes immer bei den Rechnungen – das Bit für andere Zwecke verwenden: Also als Vorzeichenbit. Taucht hier also eine 0 auf, dann liegt eine positive Zahl vor, ist es aber eine 1, dann signalisiert diese eine negative Zahl. Für das MFLPT-Format muß in unserem Beispiel also Bit 7 des ersten Mantissenbytes gelöscht werden (1985,125 ist ja nun mal positiv) und die komplette Zahl sieht im MFLPT-Format so aus:

1000 1011	0111 1000	0010 0100	0000 0000	0000 0000
Byte 1	↑ Byte 2	Byte 3	Byte 4	Byte 5
Exponent	M A	N T	I S	S E

Der Pfeil weist auf das Vorzeichenbit. Man spricht hier auch vom »gepackten« Format. Damit das alles nun nicht nur graue Theorie bleibt und Sie auch aus eigenem Erleben diese Zahlenformate sehen können, wollen wir hier ein kleines Testprogramm ausprobieren. Es wird Ihnen auch später noch gute Dienste leisten können, wenn Sie mal irgendwelche Zahlen in das FLPT- (also FAC oder ARG) oder ins MFLPT-Format umrechnen müssen. Zu Fuß ist das ja – wie Sie nun wissen – ganz schön haarig! Wie so oft, besteht auch dieses Programm aus einem Basic-Teil, der die Benutzerführung übernimmt und zwei kleinen Maschinenroutinen, die per USR-Vektor angesprungen werden. In diesen Assembler-Programmteilen sind zwei Interpreter-Routinen verborgen, die sehr nützlich und daher erklärenswert sind. Als Bild 23 ist das Basic-Aufrufprogramm abgedruckt.

Es fragt zunächst mal, ob der SMON eingeladen ist. Der wird nämlich aus dem Programm heraus angesprungen. Wird die Frage mit »J« beantwortet, dann zeigt sich ein kleines Menü, andernfalls ist das Programm beendet: Der SMON muß erst eingeladen werden.

Das Menü bietet 3 Optionen: Eine Zahl kann im FAC (Option 1), im ARG (Option 2) oder im MFLPT-Format ab Speicherstelle $6800 (Option 3) betrachtet werden.

Für Option 1 wird der USR-Vektor auf die Einsprungadresse des SMON gestellt und dann mittels USR-

	$ / dez.					
FAC	$	61	62	63	64	65	66
	dez.	97	98	99	100	101	102
ARG	$	69	6A	6B	6C	6D	6E
	dez.	105	106	107	108	109	110
Inhalt Binär		1000 1011	1111 1000	0010 0100	0000 0000	0000	
Hex.		8B	F8	24	00	0000	0... ...
Dez.		139	248	36	0	0	
Byte Nr.		1	2	3	4	5	6
Erläuterung		Exponent	Mantisse				Vorzeichen

Bild 22.
So sieht die Zahl 1985,125 komplett im FAC und ARG aus

```
5 REM*** TEST FUER FLPT UND MFLPT ***         <162>
10 POKE 52,96:POKE 56,96:CLR:PRINT CHR$(14
   7)CHR$(17)CHR$(17)                         <215>
15 PRINT"IST DER SMON EINGELADEN?":INPUT"J
   /N";A$:IF A$="N"THEN END                   <254>
20 PRINT CHR$(17)CHR$(17)"{3SPACE}FLPT IN
   FAC"TAB(25)"1":PRINT                       <003>
30 PRINT"{3SPACE}FLPT IN ARG"TAB(25)"2":PR
   INT                                        <030>
40 PRINT"{3SPACE}MFLPT AB $6800"TAB(25)"3"
   :PRINT:PRINT                               <077>
50 GET A$:IF A$<"1"OR A$>"3"THEN 50           <211>
60 PRINT"AUSWAHL",A$:PRINT:PRINT:INPUT"GEB
   EN SIE EINE ZAHL EIN";Z                    <107>
65 PRINT CHR$(147)                            <142>
70 ON VAL(A$)GOTO 100,200,300                 <233>
100 REM***** FAC *****                        <097>
110 POKE 785,0:POKE 786,192:REM USR-VEKTOR
     AUF SMON = $C000                         <091>
120 A=USR(Z)                                  <205>
200 REM***** ARG *****                        <213>
210 POKE 785,0:POKE 786,96:REM USR-VEKTOR
    AUF $6000                                 <011>
220 A=USR(Z)                                  <049>
300 REM***** MFLPT *****                      <227>
310 POKE 785,0:POKE 786,97:REM USR-VEKTOR
    AUF $6100                                 <114>
320 A=USR(Z)                                  <150>
400 REM**********************************     <138>
410 REM NACH MELDUNG DES SMON MIT DEN         <042>
420 REM KOMMANDOS                             <221>
430 REM (1)    M 0061                         <212>
440 REM (2)    M 0069                         <231>
450 REM (3)    M 6800                         <241>
460 REM DEN MONITOR EINSCHALTEN. DIE          <137>
470 REM EINGEGEBENE ZAHL IST DANN ALS         <148>
480 REM HEX-BYTES SICHTBAR.                   <135>
490 REM**********************************     <228>

© 64'er
```

Bild 23. Testprogramm für die beiden kleinen Assembler-Routinen. Die Bedienung ist im Artikel erklärt.

die Zahl Z in den FAC übergeben. Es schaltet sich dann der SMON ein, der nun mittels des Kommandos M 0061 den Inhalt des FAC als Hex-Zahlen zeigt.

Option 2 richtet zunächst den USR-Vektor auf ein kleines Assembler-Programm ab $6000, welches den FAC-Inhalt in den ARG schiebt, dann den USR-Vektor auf den SMON richtet und schließlich auch diesen einschaltet. Auch hier wird mit dem M-Kommando dann per M 0069 der ARG-Inhalt sichtbar. Option 3 richtet den USR-Vektor auf eine Maschinenroutine, die bei $6100 beginnt. Dort wird der FAC-Inhalt nach $6800 und folgende Speicherstellen verschoben und zwar ins MFLPT-Format. Anschließend erfolgt dann wieder das Ausrichten des USR-Vektors auf den SMON, Anschalten des SMON, wo man durch M 68000C den Inhalt ansehen kann. Folgende Vorgehensweise empfehle ich Ihnen:

1. Einladen des SMON

2. Eintippen der beiden kleinen Assembler-Routinen mit Hilfe des SMON und Speichern (man kann einfach mit dem SMON-Kommando S→Programmname→,6000610A speichern).

2a. Wenn Sie die beiden Routinen schon gespeichert vorliegen haben, dann laden Sie sie jetzt ein. Jedenfalls sollten Sie nach dem Laden beider Assembler-Programme (SMON und die beiden Routinen) ein NEW eingeben, so daß alle Zeiger zurückgestellt werden.

3. Erst jetzt Laden oder Eintippen des Basic-Aufrufprogrammes.

Wenn Sie nun das Testprogramm starten und zum Beispiel unsere Zahl 1985,125 eingeben, werden Sie folgendes finden:

```
Option 1:
M0061
:0061 8B F8 24 00 00 78 00 00

Option 2:
M0069
:0069 8B F8 24 00 00 78 D4 CE

Option 3:
M6800
:6800 8B 78 24 00 00 FF FF FF
```

Die Bytes, welche zu unserer Zahl gehören, sind unterstrichen. Sie können jeweils nach RUN/STOP noch mit dem SMON-Kommando $8B (oder eine andere Sie interessierende Hexzahl) eine Ausgabe im Binär- und im Dezimalformat erreichen.

So, nun aber endlich zu den beiden Assembler-Routinen. Zur Option 2 gehört das folgende, bei $6000 beginnende Programm:

```
6000   JSR   BC0C
```

$BC0C ist die erste Interpreter-Routine, die wir uns zunutze machen. Sie schiebt den Inhalt vom FAC in den ARG. Mehr dazu später.

```
6003   LDA   #00
6005   STA   0311
6008   LDA   #C0
600A   STA   0312
```

Damit haben wir den USR-Vektor auf $C000 gestellt.

```
600D   JMP   C000
```

Das war das Einschalten des SMON. Im Grunde genommen könnten wir uns das Stellen des USR-Vektors ersparen.

Es ist aber sinnvoll – vor allem bei langen Programmen – wenn verstellte Vektoren nach Beendigung des Programmes auf einem definierten Wert stehen.

Nun noch die Routine für Option 3:

```
6100   LDX   #00
6102   LDY   #68
6104   JSR   BBD4
```

Auch das ist wieder eine Interpreter-Routine: Sie schiebt den FAC-Inhalt in einen Speicherbereich, dessen Startbyte durch die beiden Index-Register angegeben wird (X-Register für LSB, Y-Register für MSB, hier also 6800). Dabei wird die Zahl vom FLPT-Format in das MFLPT-Format umgewandelt. Das Programmchen schließen wir ab mit einem Sprung zum Rest der ersten Routine:

```
6107   JMP   6003
```

Sehen Sie sich mal einige Zahlen im Fließkomma-Format an. Fast alle Operationen mit Zahlen vollführt unser Computer mit diesen Fließkommazahlen. Das ist dann beispielsweise der Grund dafür, daß aus einer Basic-Zeile wie der folgenden:

```
IF INT(X*10)=INT(ABS(X*10))THEN ...
```

auch bei positiven X-Werten (wo man mathematisch Gleichheit feststellt) manchmal die Bedingung als nicht erfüllt erkannt wird. X wird sofort als Fließkommazahl in den FAC gelegt, mit einer Fließkomma-Zehn multipliziert, der ABS-Wert wird ebenfalls per Fließkomma-Arithmetik ermittelt und so weiter. Dabei treten häufig Rundungsprobleme auf, wenn ein Zwischenergebnis mehr als 32 signifikante binäre Nachkommastellen aufweist (wie wir es ja zum Beispiel beim periodischen Binärbruch gesehen haben, der sich aus der simplen Dezimalzahl 0,1 ergibt). Das Rechnen mit Fließkommazahlen im Computer öffnet zwar einen ungeheuren Zahlenraum für unsere Anwendungen, es geht aber viel langsamer als die 2-Byte-Arithmetik. Immerhin müssen hier jedesmal 6 Byte (beziehungsweise 5 bei MFLPT) berücksichtigt werden. Ich glaube aber kaum, daß wir jemals in die Verlegenheit kommen werden, beispielsweise eine Fließkomma-Addition programmieren zu müssen. Eben weil unser C 64 fast alle Zahlenoperationen mit Fließkomma-Formaten durchführt, sind nahezu alle Eventualitäten schon als fertige abrufbare Programme im Interpreter enthalten. Wir müssen nur wissen, wie unsere Zahlen aussehen (das haben Sie nun ja gelernt) und wo und wie man sie für Operationen bereithält und wo und wie man die entsprechenden Routinen finden kann. Einen der wichtigsten Wege, unsere Zahlen ans Maschinenprogramm zu übergeben, haben Sie schon kennengelernt: Das Argument der USR-Funktion landet automatisch im FLPT-Format im FAC.

39. Die beiden ersten Interpreter-Routinen

Von nun an sollen nach und nach Interpreter-Routinen vorgestellt werden. Das ist allerdings nicht so einfach wie bei der Kernel-Sprungtabelle. Es gibt für die letzteren viele recht gut dokumentierte Listen. Für die Interpreter-Routinen ist kaum Literatur vorhanden. Will man die ähnlich erfassen wie die Kernel-Routinen, dann muß man ROM-Listings wälzen und vor allem probieren, probieren ... Falls Sie also mal einen Fehler in der Beschreibung feststellen oder Dinge, die ich leer lassen muß, weil mir dazu die Erleuchtung noch nicht gekommen ist, selbst schon kennen, dann schreiben Sie mir. Gemeinsam haben wir vielleicht die Chance, auch die letzte im Interpreter versteckte Nuß noch zu knacken!

Nun also zur ersten schon verwendeten Routine:

Name	MOVAF
Zweck	Übertragen des FAC in den ARG
Adresse	$BC0C, dez. 48140
Vorbereitung	Wert in FAC
Speicherstellen	$61-66 FAC, $69-6E ARG, $6F, $70
Register	Akku, X-Register
Stapelbedarf	4

Diese Routine ist deswegen so wichtig, weil viele Rechenoperationen, die zwei Zahlen verknüpfen, zwischen dem FAC und dem ARG abgewickelt werden. Wenn Sie unser kleines Testprogramm mal mit der Option 2 laufen lassen und hinterher nicht nur mit M0069 in den ARG, sondern auch mit M0061 in den FAC hineinsehen, dann stellen Sie fest, daß der FAC-Inhalt noch immer vorhanden ist.

Allerdings ist das nicht immer der Fall. MOVAF rundet nämlich – wenn nötig – vorher noch den FAC-Inhalt, der dann natürlich anders aussieht.

Fast noch häufiger benutzt man die zweite Interpreter-Routine:

Name	MOVMF
Zweck	Übertragung von FAC in Speicher unter Umrechnung ins MFLPT-Format
Adresse	$BBD4 dez. 48084
Vorbereitung	Wert in FAC, Zieladresse in X- und Y-Register (X = LSB, Y = MSB)
Speicherstellen	$61-66 FAC, $70, $22, $23
Register	Akku, X- und Y-Register
Stapelbedarf	4

Außer den unter »Speicherstellen« genannten sind natürlich auch noch die Zieladresse und deren vier nachfolgende Bytes in die Routine einbezogen (das MFLPT-Format besteht ja aus 5 Byte). $22/23 ist ein für die Operation verwendeter Zeiger.

MOVMF wird häufig dann verwendet, wenn Werte aus welchen Gründen auch immer, außerhalb der Fließkomma-Akkumulatoren gelagert werden müssen.

Es wird Ihnen vielleicht aufgefallen sein, daß im Gegensatz zur Beschreibung der Kernel-Routinen – die Rubrik »Fehler« fehlt. Der Grund ist, daß es keine solchen Sicherungen bei den Interpreter-Routinen gibt. Was passieren kann, ist unter bestimmten Bedingungen das Ansteuern von normalen Basic-Fehlermeldungen, die aber nicht immer den tatsächlichen Zustand wiedergeben. Wenn Ihnen mal bei der Programmierung mit Interpreter-Routinen Zweifel aufkommen, dann verfolgen Sie lieber den Programmweg mittels eines ROM-Listings und schalten Sie eigene Fehler-Routinen ein. Das war aber nur für die Fortgeschrittenen gesagt. Wir werden uns erst nach und nach dahin vortasten. Zunächst fehlen uns ja noch ein paar Assembler-Kenntnisse. Mit dem nächsten Abschnitt soll das besser werden.

40. Assembler-Befehle zum Beherrschen von Bits

Fangen wir also mit AND an. AND verknüpft den Akku-Inhalt Bit für Bit mit dem angegebenen Wert nach den Regeln der logischen UND-Verknüpfung. Die Adressiermöglichkeiten dieses Befehls sind allerlei:

AND 6000	absolut
AND FE	Zeropage absolut
AND #07	unmittelbar
AND 6000,X	absolut-X-indiziert
AND 6000,Y	absolut-Y-indiziert
AND (FA,X)	indiziert-indirekt
AND (FB),Y	indirekt-indiziert
AND FE,X	Zeropage-absolut-X-indiziert

Damit haben wir eine ganze Menge an Möglichkeiten. Erinnern Sie sich noch an die Regeln einer UND-Verknüpfung? Wenn nicht, dann sehen Sie sich nochmal die Tabelle 12 an.

Sie erkennen, daß zwei miteinander AND-verknüpfte Bits nur dann als Ergebnis 1 haben, wenn in beiden Bits der Wert 1 steht. Man kann mittels AND ganz gezielt Bits löschen. Nehmen wir mal als Beispiel an, wir wollten geshiftete Zeichen (das sind die mit den Codes größer als 128) in normale Zeichen umwandeln. Dazu bringen wir die Zeichencodes in den Akku und löschen Bit 7. Übrig bleibt dann der Code für das ungeshiftete Zeichen. Für das Löschen von Bit 7 brauchen wir eine sogenannte UND-Maske, die dafür sorgt, daß alle anderen Bits unverändert bleiben. An den Stellen muß in dieser Maske also eine 1 stehen (denn 0 AND 1 ergibt 0, 1 AND 1 ergibt 1). Lediglich Bit 7 der Maske muß 0 sein. Die Maske muß also heißen:

0111 1111 $7F dez. 127

Nehmen wir an, im Akku befände sich der Code für ein geshiftetes A, also dez. 193 (binär 1100 0001, $C1), dann ergibt die AND-Verknüpfung mit der Maske:

Akku	1100	0001	Shift A
Maske	0111	1111	
AND			
Jetzt im Akku	0100	0001	

Normales A (Code dez. 65, $41)

Man kann also, je nach Wahl der Maske, beliebige Bits löschen.

AND ist, je nach der gewählten Adressierungsart, ein 2- oder 3-Byte-Befehl. Weil das Ergebnis im Akku steht, können Flaggen beeinflußt werden. Die N- und die Z-Flagge reagieren auf das Ergebnis.

Im Gegensatz zu Basic, wo es nur eine ODER-Verknüpfung gibt, nämlich OR, existieren im Assembler zwei davon. Man unterscheidet ein »inklusives« und ein »exklusives« ODER. Die inklusive ODER-Verknüpfung des Akku mit den angegebenen Daten geschieht mit dem Assembler-Befehl ORA. ORA entspricht dem Basic-Befehl OR. Alle Adressierungsarten, die dem AND-Befehl offenstehen, können auch auf ORA angewendet werden. Wenn man Bits ORA-verknüpft, findet man folgende Ergebnisse:

0 ORA 0 = 0
0 ORA 1 = 1
1 ORA 0 = 1
1 ORA 1 = 1

Auch hier ist eine sogenannte Wahrheitstabelle recht einprägsam (siehe Tabelle 13).

Während man mit AND gezielt Bits löschen kann, ist es mit ORA möglich, Bits zu setzen. Auch dazu verwendet man eine Maske, die an allen Stellen, an denen Bits unverändert bleiben sollen, eine 0, sonst aber eine 1 enthält. Nehmen wir nochmal das Beispiel von vorhin und wandeln nun das ungeshiftete Zeichen in ein geshiftetes um. Wir müssen also Bit 7 wieder setzen: Da muß in der Maske dann eine 1 stehen. Alle anderen Bits bleiben unverändert, wenn die Maske dort eine Null aufweist. Die Maske muß daher heißen:

1000 0000 $80 dez. 128

Im Akku soll das ungeshiftete B stehen (Code dez. 66, $42, bin. 0100 0010). Die Rechnung sieht dann so aus:

Akku	0100	0010	Code für B
Maske	1000	0000	
ORA			
Jetzt im Akku	1100	0010	

AND	0	1
0	0	0
1	0	1

Tabelle 12. Wahrheitstabelle zur AND-Verknüpfung

ORA	0	1
0	0	1
1	1	1

Tabelle 13. Wahrheitstabelle zur ORA-Verknüpfung

Code für geshiftetes B.

Je nach Art der Maske kann man also ein oder mehrere Bits setzen. Im Beispiel ist auch der Einfluß dieses Befehls auf die Flaggen zu erkennen. Der Akku-Inhalt vor der ORA-Operation hatte kein Bit 7, also keine gesetzte N-Flagge. Danach ist Bit 7 gesetzt und die N-Flagge zeigt eine 1. Außer der N-Flagge kann – ebenso wie beim AND-Befehl – auch noch die Z-Flagge reagieren. ORA ist je nach Adressierungsart ein 2- oder 3-Byte-Befehl.

Während zwei Bit in der ORA-Verknüpfung eine 1 ergeben, wenn sie beide gesetzt sind oder eines von beiden, schließt die EOR-Verknüpfung den ersten Fall aus. EOR ist die exklusive ODER-Verknüpfung. Sie läßt sich sprachlich erfassen im »entweder ... oder ...«, also beispielsweise: Beim Roulette fällt die Kugel entweder auf Rouge oder auf Noir, beides zusammen ist nicht möglich. Die Regeln bei EOR sind also:

0 EOR 0 = 0
0 EOR 1 = 1
1 EOR 0 = 1
1 EOR 1 = 0

Eine Wahrheitstabelle dazu sehen Sie in Tabelle 14.

Wozu verwendet man EOR? Es fällt Ihnen vielleicht auf, daß wir die aus Basic bekannte NOT-Funktion nicht in Assembler vorliegen haben. Obwohl EOR einige viel weitergehendere Verwendungsmöglichkeiten aufweist als NOT (aber auf Boolesche Algebra wollen wir hier nicht eingehen), kann man es mit gleicher Wirkung einsetzen. Wir haben beispielsweise in den ersten Kapiteln negative Zahlen durch Komplementieren erzeugt. Dabei sollte jedes Bit in sein Gegenteil verkehrt werden. Das wäre die Aufgabe einer NOT-Funktion. Durch ein EOR FF können wir dasselbe erreichen. Sehen wir uns wieder ein Beispiel an. Im Akku steht dez. 15 ($0F, bin. 0000 1111):

```
Akku          0000 1111
Maske         1111 1111   =$FF
 EOR
              ---------------------
Jetzt im Akku 1111 0000
```

Einerkomplement von dez. 15.

Auch EOR kann alle Adressierungsarten verkraften, die die beiden anderen logischen Assembler-Befehle erlauben. Je nach der gewählten Art liegt dann ein 2- oder 3-Byte-Befehl vor. Auch hier werden die Z- und die N-Flagge beeinflußt.

Das waren also die logischen Befehle. Leider ist hier nicht der geeignete Ort, die Vielseitigkeit, die damit möglich ist, deutlich zu machen. Wenn Sie sich dafür interessieren, sollten Sie mal etwas über Boolesche Algebra lesen oder eine Einführung in die mathematische Logik.

Um dieses Thema abzuschließen, soll noch erwähnt werden, daß der Basic-Interpreter so eingerichtet ist, daß er immer dann, wenn die Richtigkeit einer Aussage zu überprüfen ist, mit -1 antwortet bei wahrer Aussage, dagegen mit 0 bei falscher. Auf diese Weise kommen diese merkwürdigen Basic-Programmzeilen ins rechte Licht, in denen Sequenzen auftauchen wie:

```
C=A-161-33*(A< 255)-64*(A < 192)-32*(A <160)+
32*(A < 96)-64*(A < 64).
```

Jedesmal, wenn zum Beispiel A < 64 ist, tritt anstelle der Klammer ein -1 auf. Übrigens ist diese Formel eine schöne kurze Möglichkeit, ASCII-Code (hier A als Variable) in den Bildschirmcode umzurechnen (der Bildschirmcode steht dann in der Variablen C).

EOR	0	1
0	0	1
1	1	0

Tabelle 14. Wahrheitstabelle zur EOR-Verknüpfung

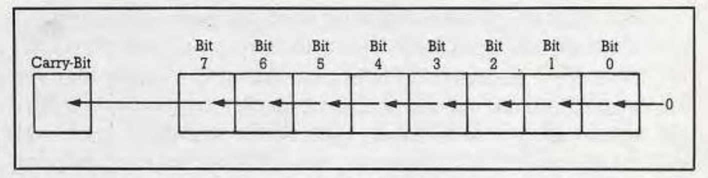


Bild 24. Wirkung des ASL-Befehls: Arithmetisches Linksschieben

Kommen wir nun zur zweiten Gruppe von Assembler-Befehlen, die Bit-Manipulationen erlauben: den Verschiebe-Befehlen. Fangen wir dabei mit ASL an, was vom englischen »Arithmetic Shift Left« kommt. Zu deutsch heißt das dann »arithmetisches nach links schieben«. Davon sind wir aber auch noch nicht schlauer. Sehen wir uns an, was dieser Befehl tut (Bild 24).

Der gesamte Inhalt des Akku beziehungsweise der Speicherstelle (je nach Adressierung) wird um eine Bit-Position nach links verschoben. Das vorherige Bit 7 wandert in die Carry-Flagge, alle anderen Bits erhalten eine um 1 höhere Position, das freigewordene Bit 0 wird mit einer 0 aufgefüllt. Toll! Aber was soll das? Zur Erklärung machen wir nochmal einen kurzen Ausflug zu unserem normalen dezimalen Zahlensystem. Nehmen wir mal die Zahl 123. Bei der Einführung in die Fließkommazahlen haben wir gelernt das Komma zu verschieben. 123 ist ja dasselbe wie 123,00. Wenn wir das Komma um eine Stelle nach rechts verschieben, erhalten wir 1230,0 (dabei lassen wir jetzt mal den Exponenten außer acht, der wäre ja -1, weil 123,00 = $1230{,}0 \times 10^{-1}$). Man kann das Ganze auch andersherum sehen: Wir haben die Zahl 123 eine Stelle nach links verschoben und die freigewordene Stelle ganz rechts mit einer Null aufgefüllt. 1230,0 ist das Zehnfache von 123,00. Die Verschiebung um eine Stelle nach links hat also zur Multiplikation unserer Zahl mit der Basis unseres Zahlensystems (also 10) geführt. Eine zweimalige Linksverschiebung führt zu 12300, den 100fachen Wert unserer Ausgangszahl. Wir haben also die Zahl 123,00 mal 10 mal 10 genommen, das sind 10^{-2}. Jede Linksverschiebung erhöht unseren Ausgangswert um eine Zehnerpotenz, oder – anders ausgedrückt – erhöht den Multiplikator um eine Zehnerpotenz und deshalb natürlich auch das Ergebnis (einmal linksschieben: Multiplikator = 10 = 10^{-4}, zweimal linksschieben: Multiplikator: 100 = 10^{-2} und so weiter).

Im Binärsystem, zu dem wir nun wieder zurückkehren, ist die Zahlenbasis die Zahl 2. Einmal Linksschieben entspricht dann einer Multiplikation mit $2^{-1}=2$. Das zweimalige Linksschieben führt zur Multiplikation mit $2^{-2}=4$ und so weiter. Nehmen wir als Beispiel die Zahl 3, welche im Binärsystem 0000 0011 heißt:

1. ASL	führt zu	0000 0110	= dez.6 (2^1x3=2x3=6)
2. ASL		0000 1100	= dez. 12 (2x6= 12, 2^2x3=4x3=12)
3. ASL		0001 1000	= dez.24 (2x12=24, 2^3x3=8x3=24)
4. ASL		0011 0000	= dez. 48 (2x24=48, 2^4x3=16x3=48)
5. ASL		0110 0000	= dez. 96 (2x48=96, 2^5x3=32x3=96)
6. ASL		1100 0000	= dez. 192 (2x96=192, 2^6x3=63x3+192)

Bis jetzt landete im Carry-Bit immer eine Null. Wenn wir nun nochmal linksschieben, finden wir darin eine 1, die offensichtlich als Bit 8 unseres Ergebnisses dienen muß:

7. ASL (1) 1000 0000 = (mit Carry als Bit 8) dez. 384 (2x192=384, 2^{-7}x3= 128x3=384

Daraus folgt, daß immer dann, wenn man sich nicht hundertprozentig sicher ist, eine Abfrage des Carry-Bits erfolgen sollte, sofern man ASL zum Rechnen einsetzt (BCC beziehungsweise BCS bieten sich da an). Dazu kommen wir noch. Sehen wir uns zunächst mal an, wie ASL adressierbar ist:

ASL

ohne Adresse, der Akkuinhalt wird nach links verschoben.

Manchmal als eigene Adressierungsart bezeichnet.

ASL 6000	absolut
ASL FE	Zeropage-absolut
ASL 6000,X	absolut-X-indiziert
ASL FA,X	Zeropage-absolut-X-indiziert

Je nach Adressierung tritt ASL dann als 1-, 2- oder 3-Byte-Befehl auf. Die N-, die Z- und die Carry-Flagge werden beeinflußt. Das Ergebnis steht bei der ersten Adressierungsart (also ASL ohne Adresse) im Akku. In den anderen Fällen findet man es in der jeweiligen Speicherstelle.

Nun gut, werden Sie sagen, man kann also mittels ASL Zahlen mit 2, 4, 8, 16 32 etc. multiplizieren. Was aber, wenn man mal 40 nehmen will? Da gibt es einige Möglichkeiten, die ein bißchen den Erfindungsgeist ansprechen. Man kann ja, wenn irgendeine Zahl Z mal 40 gerechnet werden soll, dafür schreiben:

40*Z=(32+8)*Z=32*Z+8*Z

Schon haben wir wieder Multiplikatoren, die den Einsatz von ASL ermöglichen. Die beiden Zwischenergebnisse (als 32xZ und 8xZ) speichern wir irgendwo ab und zählen sie dann zusammen. Wenn Z zum Beispiel 3 wäre, könnte man das so programmieren:

```
6000    STA    6100
```

Dabei sollte im Akku Z also die 3 stehen, die wir nun zwischengespeichert haben.

```
6003    ASL
6004    ASL
6005    ASL
6006    ASL
6007    ASL
```

Jetzt liegt im Akku der 32fache Wert von 3, also 96 vor und wir speichern dieses Zwischenergebnis ab.

```
6008    STA    6101
600B    LDA    6100
```

Wir haben nun den Wert 3 aus dem Zwischenspeicher $6100 wieder in den Akku geholt und schieben ihn 3mal nach links um den 8fachen Wert zu erhalten.

```
600E    ASL
600F    ASL
6010    ASL
```

Nun erfolgt das Zusammenzählen beider Zwischenergebnisse. Dabei ist ja 8xZ noch im Akku.

```
6011    CLC
6012    ADC    6101
```

Damit ist die Aufgabe gelöst. Das Ergebnis steht im Akku und kann nun weiter verwendet werden.

Auf diese Weise kann man immer einen Mulitplikator in eine Zweierpotenz (2, 4, 8, 16,...) und weitere Summanden zerlegen. Dies ist allerdings eine zwar schnelle, aber doch recht eingeschränkte Art der Multiplikation. Außerdem haben Sie noch nicht erfahren, wohin man denn nun am besten mit BCC verzweigt, wenn die 8 Bit des Ergebnisses überlaufen.

Abschließend finden Sie in Tabelle 15 noch alles Wissenswerte zu den neuen Befehlen.

41. Die restlichen Bit-Verschiebe-Operationen

Da wäre zunächst einmal das Gegenstück zu ASL. Dort ging es ja um das nach links schieben. Jetzt schieben wir nach rechts. LSR heißt der dazu nötige Befehl. Das kommt von »logical shift right« und heißt zu deutsch »logisches Rechtsschieben«. Fragen Sie mich bitte nicht, weshalb »logisches«. Jedenfalls ist LSR ebenso für logische Bitprüfungen geeignet wie ASL.

Mittels LSR wird jedes Bit der adressierten Speicherstelle um einen Platz nach rechts geschoben. An die Stelle des Bit 7 tritt eine Null und Bit 0 wandert in das Carry-Bit (siehe Bild 25).

Erinnern Sie sich noch an das dezimale Linksschieben mit ASL? Wir hatten festgestellt, daß jedes Linksschieben einer Dezimalzahl einer Multiplikation mit 10 entspricht. Hier im umgekehrten Fall, also beim Rechtsschieben, muß jedes LSR einer Division durch 10 entsprechen:

25000	wird durch LSR	2500
2500	zu "	250
250	"	25

und so weiter

Geht man von der Ausgangszahl (25000) aus, dann ergibt sich der erste rechts verschobene Wert durch Division mit

	$10^1=10$
der 2. durch	$10^2=100$
der 3. durch	$10^3=1000$, etc.

Es wird also durch Potenzen der Zahlenbasis 10 geteilt. Haben wir es – wie im Computer – mit Binärzahlen zu tun, deren Basis die 2 ist, dann teilen wir mit jedem LSR durch 2. Je nachdem, wie oft hintereinander das LSR auf eine Zahl ausgeübt wird, teilt man dann durch $2^1=2$, $2^2=4$, $2^3=8$, etc.

Befehlswort	Adressierung	Bytezahl	Code Hex	Code Dez	Taktzyklen	Beeinflussung von Flaggen
AND	absolut	3	2D	45	4	N, Z
	0-page-abs	2	25	37	3	N, Z
	unmittelbar	2	29	41	2	N, Z
	abs.-X-indiz.	3	3D	61	4 *	N, Z
	abs.-Y-indiz.	3	39	57	4 *	N, Z
	indiz.-indir.	2	21	33	6	N, Z
	indir.-indiz.	2	31	49	5 *	N, Z
	0-page-X-indiz	2	35	53	4	N, Z
ORA	absolut	3	0D	13	4	N, Z
	0-page-abs.	2	05	05	3	N, Z
	unmittelbar	2	09	09	2	N, Z
	abs.-X-indiz.	3	1D	29	4 *	N, Z
	abs.-Y-indiz.	3	19	25	4 *	N, Z
	indiz.-indir.	2	01	01	6	N, Z
	indir.-indiz.	2	11	17	5 *	N, Z
	0-page-X-indiz	2	15	21	4	N, Z
EOR	absolut	3	4D	77	4	N, Z
	0-page abs.	2	45	69	3	N, Z
	unmittelbar	2	49	73	2	N, Z
	abs.-X-indiz.	3	5D	93	4 *	N, Z
	abs.-Y-indiz.	3	59	89	4 *	N, Z
	indiz.-indir.	2	41	65	6	N, Z
	indir.-indiz.	2	51	81	5 *	N, Z
	0-page-X-indiz	2	55	85	4	N, Z
ASL	»Akkumulator«	1	0A	10	2	N, Z, C
	absolut	3	0E	14	6	N, Z, C
	0-page-abs.	2	06	06	5	N, Z, C
	abs.-X-indiz.	3	1E	30	7	N, Z, C
	0-page-X-indiz	2	16	22	6	N, Z, C

* bedeutet: Bei seitenüberschreitenden Indizierungen muß noch ein Taktzyklus dazugerechnet werden.

Tabelle 15. Alles Wissenswerte der neuen Assembler-Befehle

Das konnte man sich alles ja schon vorstellen, nachdem ASL zur Multiplikation verwendet wurde. Auch hier muß man immer das Carry-Bit abfragen, denn die Division kann ja unter Umständen nicht aufgehen, wie das folgende Beispiel der Division von 3 durch 2 zeigt:

(3) 0000 0011 ergibt durch LSR: 0000 0001 und 1 im Carry-Bit. Das Ergebnis ist schon richtig, nämlich 1. Im Carry steht der Rest dieser Division, die 1. Weil der Rest für manche Berechnungen von Bedeutung ist, muß das Carry-Bit irgendwie erfaßt werden. Wie man das erreicht, lernen wir später noch. Leider ist diese Art der Division mittels LSR nicht so einfach verwendbar wie die Multiplikation mittels ASL. Während man dort durch geschicktes Aufteilen des Faktors ASL auch bei anderen Multiplikatoren als reine Zweierpotenzen anwenden konnte, ist das hier nicht so ohne weiteres möglich. Bei Divisionen geht man deshalb lieber andere Wege. Die zeige ich Ihnen ebenfalls etwas später.

LSR kann auf die gleiche Weise adressiert werden wie ASL:

LSR	auf den Akku bezogen
LSR 6000	absolut
LSR FE	Zeropage-absolut
LSR 6000,X	absolut-X-indiziert
LSR FA,X	Zeropage-absolut-X-indiziert

Im ersten Fall steht das Ergebnis im Akku, in den anderen Fällen in der jeweils adressierten Speicherstelle. Außer der N-Flagge, die in jedem Fall 0 wird, beeinflußt LSR auch die Carry-Flagge und unter Umständen die Z-Flagge. Je nach Adressierungsart liegt LSR als 1-Byte-, 2-Byte- oder 3-Byte-Befehl vor.

Sowohl bei ASL als auch bei LSR hatten wir festgestellt, daß man herausgeschobene Bits, falls sie noch von Bedeutung sind, irgendwie aus dem Carry-Bit (dort sind sie ja gelandet) an einen sinnvollen Ort schaffen muß. Das ist natürlich möglich über eine Befehlskette, in der zunächst das Carry-Bit abgefragt wird:
zum Beispiel:

```
6000    BCC 6007
6002    LDA #01
6004    STA 8000
6007    etc.
```

Wenn das Carry-Bit frei ist, wird alles weitere übersprungen. Wenn da drin etwas aufgetaucht ist, lädt man eine 1 (die ist ja im Carry-Bit) an die benötigte Speicherstelle (hier zum Beispiel 8000). Das kostet aber einige Bytes Speicherplatz und einige Taktzeiten Rechendauer. Außerdem erschwert sich die Programmierung, wenn man eine Zahl öfter verschiebt und dann nach 8000 alle Carry-Inhalte packen will. So kompliziert brauchen wir auch gar nicht zu arbeiten, denn unsere CPU kennt zwei Befehle, die das Bit-Verschieben und das Carry-Verschieben für uns machen. Das sind:

ROL rotate left	Linksrotieren
ROR rotate right	Rechtsrotieren

Sehen wir uns zunächst mal ROL (Bild 26) an:

Wie bei ASL wandert jedes Bit um eine Position nach links. Das Bit 7 wird dabei in das Carry-Bit verschoben. In Bit 0 gelangt aber hier nicht eine 0 (wie bei ASL), sondern der Inhalt des Carry-Bit (wohlgemerkt der Inhalt, der dort war, bevor dort Bit 7 hinein geschoben wurde). Bevor wir auf den praktischen Nährwert dieses Befehls eingehen, sollen erstmal die Adressierungsmöglichkeiten aufgeführt werden:

ROL		auf den Akku bezogen
ROL	6000	absolut
ROL	FE	Zeropageabsolut
ROL	6000,X	absolut-X-indiziert
ROL	FE,X	Zeropage-absolut-X-indiziert

Je nach Adressierung kann es sich dann wieder um einen 1-Byte- bis 3-Byte-Befehl handeln. Die N-, Z- und natürlich die Carry-Flagge sind beeinflußt und das Ergebnis des Befehls ist im Akku zu finden (erste Adressierungsart) oder in der angesprochenen Speicherstelle.

Wozu das Ganze? Abgesehen von der Möglichkeit, einzelne Bits auf diese Weise ohne Verlust aus einem Byte durch das Carry-Bit herausschieben zu können, um sie Prüfungen zu unterziehen, gibt es noch die Möglichkeit, einen Überlauf bei Rechenoperationen aufzufangen. Erinnern Sie sich an Kap. 40, wo wir mittels ASL Multiplikationen durchgeführt hatten? Dort kann es unter gewissen Umständen ja leicht geschehen, daß ein Byte für das Ergebnis nicht mehr ausreicht. Wir haben in den Beispielen schon die Überlegung durchgeführt, daß man mittels BCC oder BCS prüfen sollte, ob man eine signifikante Stelle (also eine führende 1) aus dem Byte herausgeschoben hat. Ist das der Fall, dann gibt es zwei Wege:

1) Man veranlaßt den Ausdruck eines OVERFLOW ERROR, wenn nur 1-Byte-Zahlen zulässig sind, oder
2) man schaltet um auf 2-Byte-Zahlen.

Sehen wir uns das mal an dem Schritt 7 des Beispiels aus Kapitel 40 an. Dort hatten wir die Zahl 192 (binär 1100 0000) vorliegen (zum Beispiel in Speicherstelle 7000). Im Computer werden 2-Byte-Integers in der Form LSB/MSB verarbeitet. Wir schaffen also die Speicherstelle für das MSB von 192 in 7001. Jetzt muß dort noch 0 drin stehen. Um bei nochmaliger Multiplikation mit 2 eine 16-Bit-Zahl als Ergebnis zu erhalten, verfährt man wie folgt:

6000 ASL 7000	Damit ist die führende 1 ins Carry-Bit gewandert
6003 BCC 6008	Das setzt man natürlich nur dann ein, wenn man nicht genau weiß, welches Ergebnis zu erwarten ist. Wenn keine 1 ins Carry-Bit gelangte, kann man die nächste Zeile überspringen.
6005 ROL 7001	Damit wurde der Inhalt des Carry-Bit als Bit 0 ins MSB unseres Ergebnisses geschoben.
6008 etc.	

Die Funktion dieser Befehlssequenz können Sie aus Bild 27 entnehmen.

Bild 25. Wirkung von LSR auf ein Byte

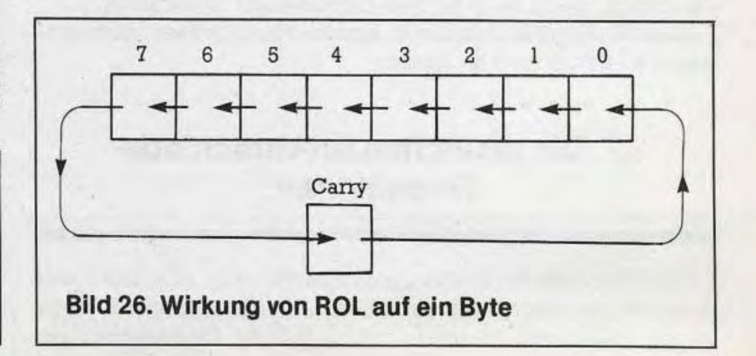


Bild 26. Wirkung von ROL auf ein Byte

Diesem Befehl werden wir später bei der 16-Bit-Multiplikation und Division noch häufig begegnen.

Sehen wir uns nun noch den letzten der Bit-Verschiebebefehle an: ROR. In Bild 28 ist schematisch gezeigt, wie rotiert wird.

Jedes Bit wandert, wie bei LSR, um eine Stelle nach rechts. Als Bit 7 kommt (im Gegensatz zu LSR) der Inhalt des Carry-Bit herein. Bit 0 wird ins Carry-Bit geschoben. Adressiert werden kann ROR ebenso wie ROL:

ROR		auf den Akku bezogen
ROR	6000	absolut
ROR	FE	Zeropage-absolut
ROR	6000,X	absolut-X-indiziert
ROR	FE,X	Zeropage-absolut-X-indiziert

Auch für die Byteanzahl, den Ort des Ergebnisses und die Flaggenbeanspruchung gilt dasselbe wie für ROL.

Die Einsatzmöglichkeiten für ROR sind allerdings geringer. Bei 16-Bit-Divisionen kann man zwar ROR einsetzen, um einen Unterlauf des MSB ins LSB aufzufangen. Weil man aber meist ohnehin andere Divisionsverfahren verwendet als das oben gezeigte mit LSR, erübrigt sich diese Anwendung in den meisten Fällen. Gut kann man ROR zu Bitprüfungen einsetzen. Das soll im nächsten Abschnitt an einem kleinen Beispiel gezeigt werden.

Zuvor aber noch eine Bemerkung: Wir sind nun durch den Befehlssatz des 6502-Assemblers fast hindurchgedrungen. Es fehlen uns nur noch – wenn ich mich nicht versehen habe – vier Befehle. Die allerdings hängen eng mit dem sogenannten Interrupthandling zusammen, das uns wohl einige Zeit beschäftigen wird.

42. Schneller Joystick

Vor einiger Zeit (64'er, Ausgabe 2/85) veröffentlichte P. Siepen eine Routine zur Abfrage des Joystickports, die eine interessante Leserbrief-Reaktion hervorrief. M. Hartig sandte nämlich einen Verbesserungsvorschlag, in dem der uns interessierende Befehl ROR die Hauptrolle spielt. Bevor ich die allerdings vorstelle, muß erst noch geklärt werden, was und wie abgefragt wird.

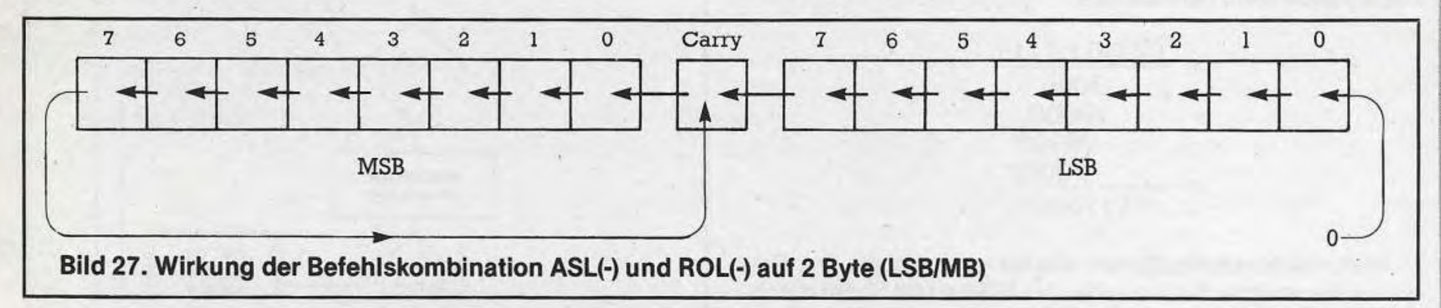


Bild 27. Wirkung der Befehlskombination ASL(-) und ROL(-) auf 2 Byte (LSB/MB)

Signale vom Joystick landen in den DATA-Ports A oder B des CIA 1. CIA heißt »Complex Interface Adapter« und ist die Institution unseres Computers, die den Verkehr mit der Außenwelt erlaubt. Wir haben zwei Stück davon (CIA 1 und CIA 2). Je nachdem, in welchen Port der Joystick gesteckt wurde, laufen die Signale in den Registern DC00 oder DC01 (dezimal 56320 oder 56321) ein. Wir nehmen im weiteren mal DC00 an. Die Bits 0 bis 4 beziehen sich auf den Joystick:

Bit 0	oben
Bit 1	unten
Bit 2	links
Bit 3	rechts
Bit 4	Feuerknopf

Wenn keine dieser Möglichkeiten angesprochen ist, enthalten diese Bits den Wert 1. Drückt man beispielsweise den Feuerknopf, dann wechselt der Inhalt von Bit 4 zum Wert 0. Man muß also ständig diese Bits überprüfen und reagieren, sobald eines davon 0 wird. Die Lösung von P. Siepen, diese Abfrage in das Interruptprogramm einzubauen, ist sehr brauchbar. Dadurch hat der Computer die Möglichkeit, trotzdem an anderen Aufgaben weiterzuarbeiten. Wir werden in den nächsten Folgen auf diese Programmiertechnik eingehen. Die Verbesserung von M. Hartig besteht darin, daß er nicht durch CMP-Befehle den Inhalt von DC00 prüft (was Zeit und auch Speicherplatz kostet), sondern mittels ROR Bit für Bit nach rechts in das Carry-Bit schiebt und dieses dann mit BCC abfragt. Sobald die Carry-Flagge nämlich frei ist, ist die zu dem Bit gehörige Joystickfunktion gefragt.

Nun die Abfrageroutine:

LDA DC00	Inhalt des DATA-Port A in den Akku
ROR	Durch Rechtsrotieren wird Bit 0 in die Carry-Flagge geschoben.
BCC Oben	Wenn die Carry-Flagge nicht gesetzt ist, war Bit 0 eine Null, also die Joystickfunktion »Oben« gefordert, zu deren Bearbeitung hier verzweigt werden kann.
ROR	Das nächste Rechtsrotieren schiebt Bit 1 in die Carry-Flagge.
BCC Unten	Auch hier wieder Abzweigen zur Bearbeitung von »Unten«, wenn Bit 1 nicht gesetzt war.
ROR	Bit 2 ins Carry-Bit
BCC Links	und bearbeiten, wenn nicht gesetzt
ROR	Bit 3 in Carry-Flagge
BCC Rechts	und verzweigen wenn Bit 3 Null war
ROR	zu guter Letzt kommt noch Bit 4 ins Carry-Bit
BCC Fire	und kann bearbeitet werden, wenn es Null war.

...weitere Bearbeitung, wenn keine Joystickfunktion

Der Vorteil dieser nur 18 Byte langen Unterroutine liegt in ihrer Schnelligkeit: Sie braucht nur 24 Taktzyklen, wenn nicht

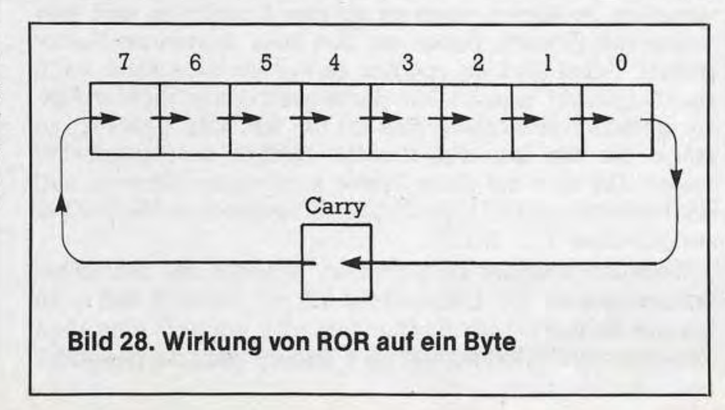


Bild 28. Wirkung von ROR auf ein Byte

verzweigt wird, beziehungsweise 25, wenn verzweigt wird. Natürlich wäre anstelle von ROR auch die Verwendung von LSR möglich gewesen, denn die herausgeschobenen Bits werden nicht mehr benötigt. Im Falle, daß man nach einer solchen Abfrage wieder den Ausgangszustand des Akku oder der Speicherstelle herstellen will, muß man eine entsprechende Anzahl ROR-Anweisungen anschließen, bis Bit 0 wieder in seine Ausgangslage rotiert ist.

43. Die 16-Bit-Multiplikation

Wir haben bisher gelernt, wie man 8-Bit-Zahlen miteinander malnehmen kann um 8- oder 16-Bit-Zahlen zu erhalten. Dabei ist unbefriedigend, daß man sich über jede Zahl Gedanken machen muß, wie man sie am besten multipliziert. Was fehlt, ist ein allgemein gültiges Programm, das in der Lage ist, jede Zahlenkombination (solange es sich um 2-Byte-Integers handelt und das Ergebnis als 16-Bit-Zahl darstellbar ist) zu verarbeiten. Und da haben wir mal wieder Glück: Gut versteckt befindet sich so etwas bereits fertig in unserem Computer. Ab dez. 45900 ($B34C) liegt im Interpreter solch eine Routine und ihr Einsprungspunkt ist für uns bei dez. 45911 ($B357). Bevor wir aber detailliert darauf eingehen, soll noch das Prinzip erklärt werden, das dabei genutzt wird.

Jeden Tag rechnen Sie wahrscheinlich völlig automatisch Multiplikationsaufgaben, ohne noch Gedanken daran zu verschwenden, wieviel Schweiß das Erlernen dieser Technik früher mal gekostet hat. Könnten Sie heute noch jemandem genau erklären, warum man da was wie macht? Genau das müssen wir aber tun, damit der Binärautomat (unser C 64) multiplizieren lernt. Nehmen wir mal eine Multiplikation von 16x15:


```
16 x 15
 16
  80
 240
```

Daß wir nicht so genau wissen, was wir da tun, liegt am ziemlich komplizierten Zehnersystem. Damit das alles einfacher und überschaubarer wird, wechseln wir mal ins Binärsystem: 16 = 10000, 15 = 1111.
Die Aufgabe sieht dann so aus:

```
10000 * 1111
 10000
  10000
   10000
    10000
 11110000
```

Jetzt wird schon deutlicher, was wir getan haben. Der Faktor auf der rechten Seite wurde vom MSB an Bit für Bit durchgesehen. Jedesmal, wenn wir auf eine 1 gestoßen sind (hier waren nur Einsen), haben wir den links stehenden Faktor notiert. Dabei sind wir von mal zu mal um eine Stelle nach rechts gerückt, was mit dem Stellenwert des im rechten Faktor gerade betrachteten Bits zu tun hat. Das geschah so lange, bis alle Bits des rechten Faktors durchgearbeitet waren. Die sich auf diese Weise ergebende Kolonne wird dann addiert und führt zum Ergebnis. Vergleichen Sie, 240 ist wirklich binär 1111 0000.

Genauso wie hier beschrieben, arbeitet das Multiplikationsprogramm. Ein Unterschied tritt auf, nämlich daß nicht bis zum Schluß mit der Addition gewartet, sondern jede neue Zwischenzahl sofort addiert wird. Bild 29 zeigt die Beschreibung der Interpreterroutine:

Name	UMULT
Zweck	Multiplikation zweier 16-Bit-Zahlen
Adresse	$B357 dez. 45911
Vorbereitungen	Faktor 1 in $28/29 Faktor 2 in $71/72
Speicherstellen	$28/29,$71/72, $5D
Register	Akku, X- und Y-Register
Stapelbedarf	keiner

Bild 29. Die Interpreterroutine UMULT

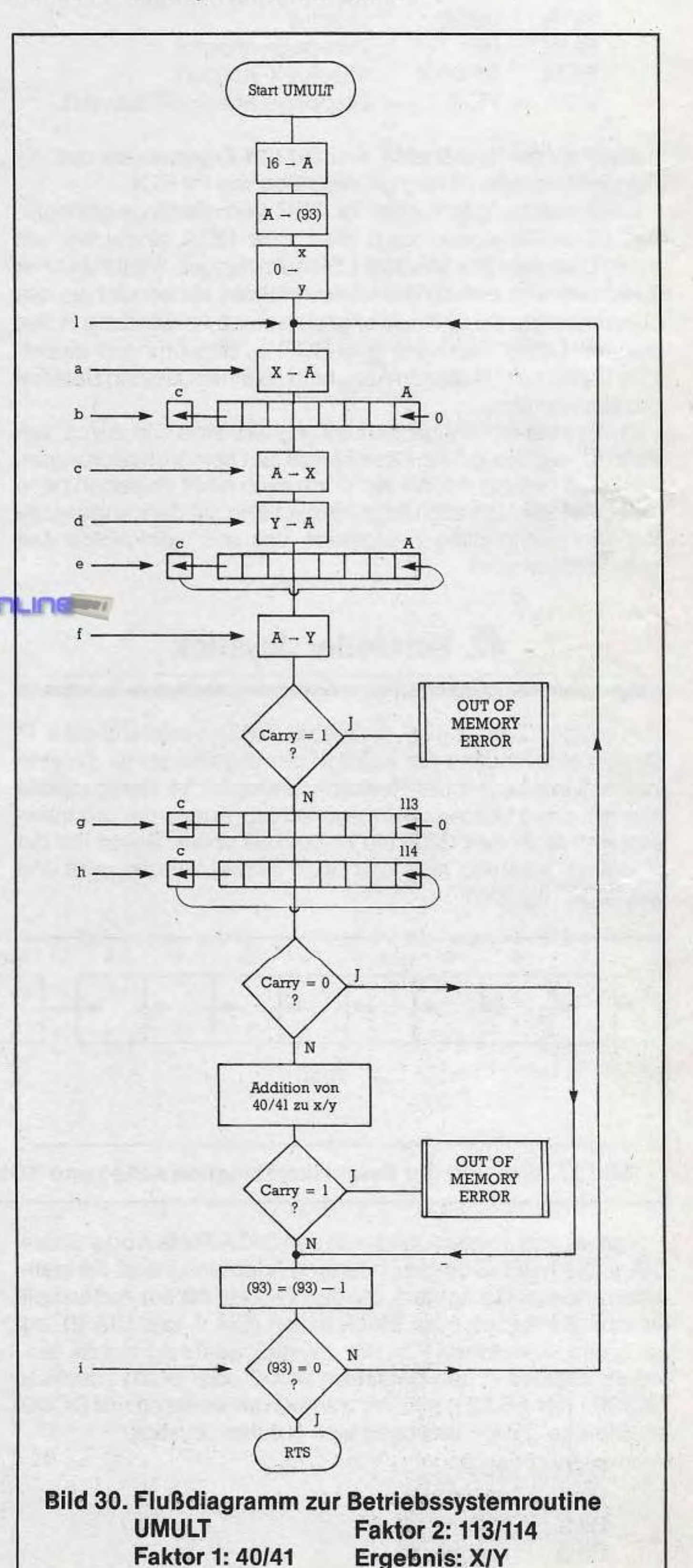


Bild 30. Flußdiagramm zur Betriebssystemroutine UMULT
Faktor 1: 40/41 **Faktor 2: 113/114** **Ergebnis: X/Y**

	114	113	Y	X	A	93	C	
I	0 0 0 0 0 0 0 0	0 1 0 0 0 0 0 1	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 1 0 0 0 0	0 0 0 1 0 0 0 0	—	Ausgangslage
a				→	0 0 0 0 0 0 0 0		—	
b					0 0 0 0 0 0 0 0		0	
c				0 0 0 0 0 0 0 0	—		0	
d				⟶	0 0 0 0 0 0 0 0		0	
e					0 0 0 0 0 0 0 0		0	
f			0 0 0 0 0 0 0 0	⟵			0	
g		1 0 0 0 0 0 1 0					0	
h	0 0 0 0 0 0 0 0						0	Ende 1.
i	0 0 0 0 0 0 0 0	1 0 0 0 0 0 1 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 1 1 1 1	0	Schleife
II	0 0 0 0 0 0 0 1	0 0 0 0 0 1 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 1 1 1 0	1, 0	Ende 2. Schleife
III	0 0 0 0 0 0 1 0	0 0 0 0 1 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 1 1 0 1	0	Ende 3. Schleife
IV	0 0 0 0 0 1 0 0	0 0 0 1 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 1 1 0 0	0	Ende 4. Schleife
V	0 0 0 0 1 0 0 0	0 0 1 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 1 0 1 1	0	Ende 5. Schleife
VI	0 0 0 1 0 0 0 0	0 1 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 1 0 1 0	0	Ende 6. Schleife
VII	0 0 1 0 0 0 0 0	1 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 1 0 0 1	0	Ende 7. Schleife
VIII	0 1 0 0 0 0 0 1	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 1 0 0 0	1, 0	Ende 8. Schleife
IX	1 0 0 0 0 0 1 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 1 1 1	0	Ende 9. Schleife
X	0 0 0 0 0 1 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 1	0 1 0 0 0 0 0 1	0 0 0 0 0 0 0 1	0 0 0 0 0 1 1 0	1, 0	Ende 10. Schleife
XI	0 0 0 0 1 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 1 0	1 0 0 0 0 0 1 0	0 0 0 0 0 0 1 0	0 0 0 0 0 1 0 1	0	Ende 11. Schleife
XII	0 0 0 1 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 1 0 1	0 0 0 0 0 1 0 0	0 0 0 0 0 1 0 1	0 0 0 0 0 1 0 0	1, 0	Ende 12. Schleife
XIII	0 0 1 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 1 0 1 0	0 0 0 0 1 0 0 0	0 0 0 0 1 0 1 0	0 0 0 0 0 0 1 1	0	Ende 13. Schleife
XIV	0 1 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 1 0 1 0 0	0 0 0 1 0 0 0 0	0 0 0 1 0 1 0 0	0 0 0 0 0 0 1 0	0	Ende 14. Schleife
XV	1 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 1 0 1 0 0 0	0 0 1 0 0 0 0 0	0 0 1 0 1 0 0 0	0 0 0 0 0 0 0 1	0	Ende 15. Schleife
XVIa,		0 0 0 0 0 0 0 0	0 1 0 1 0 0 0 0	0 1 0 0 0 0 0 0	0 1 0 1 0 0 0 0		0	
b,	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 1 0 1 0 0 0 1	1 0 0 0 0 0 0 1	0 1 0 1 0 0 0 1	0 0 0 0 0 0 0 0	1, 0	Ende 16. Schleife

Bild 31. UMULT am Beispiel der Multiplikation 321x65=20865

Diese Routine hier abzudrucken, wäre reine Platzverschwendung. Schalten Sie einfach den SMON ein und verlangen Sie von ihm ein Disassemblerlisting ab B357. Dort haben Sie dann für die weitere Besprechung alles parat. In Bild 30 finden Sie noch ein Flußdiagramm der UMULT-Routine.

Das Ergebnis der Multiplikation befindet sich in LSB/MSB-Form in den X/Y-Registern. Programm und Flußdiagramm wollen wir an einem Beispiel nachspielen. Dazu sollen die beiden Zahlen 321 und 65 (binär 0000 0001 0100 0001 und 0100 0001) miteinander multipliziert werden, was bekanntlich 20865 (binär 0101 0001 1000 0001) ergibt. Was Ihnen im Bild 31 als undurchdringlicher Bit-Dschungel entgegenstrahlt, ist das schrittweise Verfolgen des Programms im Computerformat, also binär.

In Bild 31 sind die Speicheradressen alle dezimal angegeben. Dort finden Sie zunächst die Ausgangslage. In Speicherstelle 40/41 steht die ganze Operation über unverändert die Zahl 321. In 113/114 finden Sie (wegen des LSB/MSB-Formates umgedreht als 114/113) unseren Faktor 65. Akku und Speicherstelle 93 stehen auf 16, dem Bitzähler. In das X- und Y-Register wurde eine Null eingelesen. Im Flußdiagramm ist diese Situation mit einer 1 gekennzeichnet. Ganz unten im Diagramm sehen Sie, daß der Bitzähler 93 erniedrigt und danach geprüft wird, ob er schon gleich Null sei. Daraus folgt, daß die große Schleife 16mal durchlaufen wird. Den ersten Durchlauf (gekennzeichnet durch kleine Buchstaben) verfolgen wir im einzelnen.

a) X-Register wird zur Bearbeitung in den Akku geschoben.
b) Mittels ASL wird das Bit 7 in die Carry-Flagge geschoben, was einen Carry-Inhalt von 0 bewirkt.
c) Der solchermaßen bearbeitete Akku-Inhalt (der sich hier nicht weiter verändert hat) geht wieder zurück ins X-Register.
d) Nun ist das Y-Register zur Bearbeitung dran. Es gelangt in den Akku.
e) Mittels ROL wandert nun das MSB des X-Registers aus dem Carry-Bit in die 0-Bit-Position des Akku
f) und alles zusammen wieder ins Y-Register. Insgesamt wird dadurch die 16-Bit-Zahl im X/Y-Register um eine Stellenzahl erhöht, was der Vorbereitung zur Addition dient. (Erinnern Sie sich bitte: Die Kolonne der Einzelergebnisse wird ja addiert). Im Diagramm (ohne Buchstabenkennzeichnung) schließt sich hier noch eine Prüfung auf einen eventuellen Überlauf an, der dann mit einer Fehlermeldung beantwortet wird.
g) Nun wird das MSB der Speicherstelle 113 nach links ins Carry geschoben. Das ist auch hier noch eine Null.
h) Anschließend wandert dieser Carry-Inhalt als Bit 0 in Speicherstelle 114. Bit 7 von 114 landet dafür im Carry. Auch hier wird auf diese Weise die ganze 16-Bit-Zahl 113/114 um ein Bit nach links geschoben und im nächsten Schritt – im Flußdiagramm wieder ohne Buchstabe – geprüft, ob da eine 1 oder eine 0 ins Carry-Bit geshiftet wurde. Wenn lediglich eine Null auftrat – wie hier –, dann springt das Programm sofort zum Herabzählen des Bitzählers 93. Tritt aber eine 1 auf, dann addiert sich der Inhalt von 40/41 zu X/Y.
i) Hier wird der Zustand der betroffenen Speicherstellen und Register nach dem ersten Schleifendurchlauf gezeigt.

Römisch II bis XVI zeigen nun jeweils den Zustand nach dem 2. bis 16. Durcharbeiten der großen Schleife.

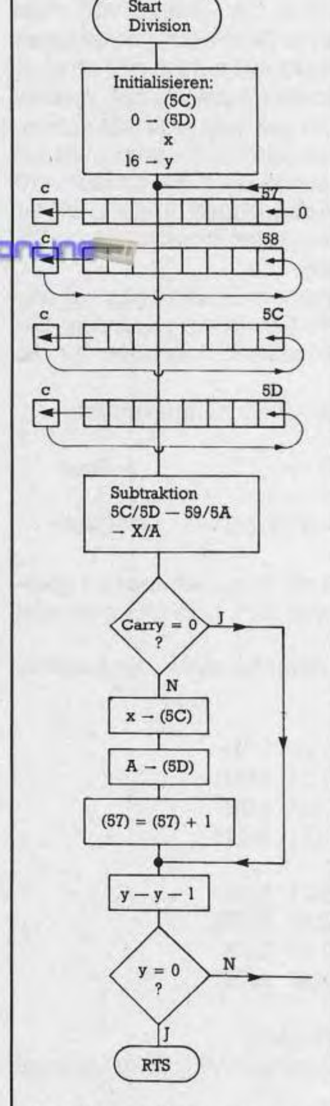


Bild 32. Flußdiagramm des Programms zur 16-Bit-Division

A : B = C + Rest
57/58 59/5a 57/58 5c/5d

```
.
,5000  A2 00     LDX #00
,5002  86 5C     STX   5C
,5004  86 5D     STX   5D
,5006  A0 10     LDY #10
,5008  06 57     ASL   57
,500A  26 58     ROL   58
,500C  26 5C     ROL   5C
,500E  26 5D     ROL   5D
,5010  38        SEC
,5011  A5 5C     LDA   5C
,5013  E5 59     SBC   59
,5015  AA        TAX
,5016  A5 5D     LDA   5D
,5018  E5 5A     SBC   5A
,501A  90 06     BCC 5022
,501C  86 5C     STX   5C
,501E  85 5D     STA   5D
,5020  E6 57     INC   57
,5022  88        DEY
,5023  D0 E3     BNE 5008
,5025  60        RTS
-----------------------------
.?
```

Programm 1. Die 16-Bit-Division

	58	57	5A	59	5D	5C	A	X	Y	C	
I a	0 1 0 1 0 0 0 1	1 0 0 0 0 0 1 1	0 0 0 0 0 0 0 1	0 1 0 0 0 0 0 1	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	—	0 0 0 0 0 0 0 0	0 0 0 1 0 0 0 0	—	Ausgangslage n. Init.
b		0 0 0 0 0 1 1 0								1	1. Linksschieben
c	1 0 1 0 0 0 1 1									0	2. Linksschieben
d, e					0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0				0	3./4. Linksschieben
f	1 0 1 0 0 0 1 1	0 0 0 0 0 1 1 0	0 0 0 0 0 0 0 1	0 1 0 0 0 0 0 1	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	1 1 1 1 1 1 1 0	1 0 1 1 1 1 1 1	0 0 0 0 1 1 1 1	1, 0	Ende der 1. Schleife
II	0 1 0 0 0 1 1 0	0 0 0 0 1 1 0 0	0 0 0 0 0 0 0 1	0 1 0 0 0 0 0 1	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 1	1 1 1 1 1 1 1 0	1 1 0 0 0 0 0 0	0 0 0 0 1 1 1 0	1, 0, 1, 0	Ende der 2. Schleife
III	1 0 0 0 1 1 0 0	0 0 0 1 1 0 0 0	0 0 0 0 0 0 0 1	0 1 0 0 0 0 0 1	0 0 0 0 0 0 0 0	0 0 0 0 0 0 1 0	1 1 1 1 1 1 1 0	1 1 0 0 0 0 0 1	0 0 0 0 1 1 0 1	1, 0	Ende der 3. Schleife
IV	0 0 0 1 1 0 0 0	0 0 1 1 0 0 0 0	0 0 0 0 0 0 0 1	0 1 0 0 0 0 0 1	0 0 0 0 0 0 0 0	0 0 0 0 0 1 0 1	1 1 1 1 1 1 1 0	1 1 0 0 0 1 0 0	0 0 0 0 1 1 0 0	1, 0, 1, 0	Ende der 4. Schleife
V	0 0 1 1 0 0 0 0	0 1 1 0 0 0 0 0	0 0 0 0 0 0 0 1	0 1 0 0 0 0 0 1	0 0 0 0 0 0 0 0	0 0 0 0 1 0 1 0	1 1 1 1 1 1 1 0	1 1 0 0 1 0 0 1	0 0 0 0 1 0 1 1	1, 0	Ende der 5. Schleife
VI	0 1 1 0 0 0 0 0	1 1 0 0 0 0 0 0	0 0 0 0 0 0 0 1	0 1 0 0 0 0 0 1	0 0 0 0 0 0 0 0	0 0 0 1 0 1 0 0	1 1 1 1 1 1 1 0	1 1 0 1 0 0 1 1	0 0 0 0 1 0 1 0	1, 0	Ende der 6. Schleife
VII	1 1 0 0 0 0 0 1	1 0 0 0 0 0 0 0	0 0 0 0 0 0 0 1	0 1 0 0 0 0 0 1	0 0 0 0 0 0 0 0	0 0 1 0 1 0 0 0	1 1 1 1 1 1 1 0	1 1 1 0 0 1 1 1	0 0 0 0 1 0 0 1	1, 0, 1, 0	Ende der 7. Schleife
VIII	1 0 0 0 0 0 1 1	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 1	0 1 0 0 0 0 0 1	0 0 0 0 0 0 0 0	0 1 0 1 0 0 0 1	1 1 1 1 1 1 1 1	0 0 0 1 0 0 0 0	0 0 0 0 1 0 0 0	1,1,0,1,0	Ende der 8. Schleife
IX	0 0 0 0 0 1 1 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 1	0 1 0 0 0 0 0 1	0 0 0 0 0 0 0 0	1 0 1 0 0 0 1 1	1 1 1 1 1 1 1 1	0 1 1 0 0 0 1 0	0 0 0 0 0 1 1 1	1, 0, 1, 0	Ende der 9. Schleife
X a	0 0 0 0 1 1 0 0	0 0 0 0 0 0 0 0			0 0 0 0 0 0 0 1	0 1 0 0 0 1 1 0	0 0 0 0 0 0 0 0	0 0 0 0 0 1 0 1		1, 0, 1, 1	
b	0 0 0 0 1 1 0 0	0 0 0 0 0 0 0 1	0 0 0 0 0 0 0 1	0 1 0 0 0 0 0 1	0 0 0 0 0 0 0 0	0 0 0 0 0 1 0 1	0 0 0 0 0 0 0 0	0 0 0 0 0 1 0 1	0 0 0 0 0 1 1 0		Ende der 10. Schleife
XI	0 0 0 1 1 0 0 0	0 0 0 0 0 0 1 0	0 0 0 0 0 0 0 1	0 1 0 0 0 0 0 1	0 0 0 0 0 0 0 0	0 0 0 0 1 0 1 0	1 1 1 1 1 1 1 0	1 1 0 0 1 0 0 1	0 0 0 0 0 1 0 1	1, 0	Ende der 11. Schleife
XII	0 0 1 1 0 0 0 0	0 0 0 0 0 1 0 0	0 0 0 0 0 0 0 1	0 1 0 0 0 0 0 1	0 0 0 0 0 0 0 0	0 0 0 1 0 1 0 0	1 1 1 1 1 1 1 0	1 1 0 1 0 0 1 1	0 0 0 0 0 1 0 0	1, 0	Ende der 12. Schleife
XIII	0 1 1 0 0 0 0 0	0 0 0 0 1 0 0 0	0 0 0 0 0 0 0 1	0 1 0 0 0 0 0 1	0 0 0 0 0 0 0 0	0 0 1 0 1 0 0 0	1 1 1 1 1 1 1 0	1 1 1 0 0 1 1 1	0 0 0 0 0 0 1 1	1, 0	Ende der 13. Schleife
XIV	1 1 0 0 0 0 0 0	0 0 0 1 0 0 0 0	0 0 0 0 0 0 0 1	0 1 0 0 0 0 0 1	0 0 0 0 0 0 0 0	0 1 0 1 0 0 0 0	1 1 1 1 1 1 1 1	0 0 0 0 1 1 1 1	0 0 0 0 0 0 1 0	1, 0	Ende der 14. Schleife
XV	1 0 0 0 0 0 0 0	0 0 1 0 0 0 0 0	0 0 0 0 0 0 0 1	0 1 0 0 0 0 0 1	0 0 0 0 0 0 0 0	1 0 1 0 0 0 0 1	1 1 1 1 1 1 1 1	0 1 1 0 0 0 0 0	0 0 0 0 0 0 0 1	1, 0, 1, 0	Ende der 15. Schleife
XVIa,	0 0 0 0 0 0 0 0	0 1 0 0 0 0 0 0			0 0 0 0 0 0 0 1	0 0 0 0 0 0 1 1	0 0 0 0 0 0 0 0	0 0 0 0 0 0 1 0		1,1,0,1,1	
b,	0 0 0 0 0 0 0 0	0 1 0 0 0 0 0 1	0 0 0 0 0 0 0 1	0 1 0 0 0 0 0 1	0 0 0 0 0 0 0 0	0 0 0 0 0 0 1 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 1 0	0 0 0 0 0 0 0 0		Ende der 16. Schleife = Endlage

Bild 33. 16-Bit-Division Schritt für Schritt am Beispiel 20867:3

Wenn Sie verstehen möchten, was da passiert, sollten Sie versuchen, Bild 31 nur als Kontrolle zu verwenden und ansonsten mal selbst alle Schritte nachzuvollziehen.

44. 16-Bit-Division

Beim umgekehrten Weg, nämlich der Teilung von zwei 16-Bit-Zahlen, haben wir nicht so viel Glück: Ich konnte keine derartige Routine im Interpreter entdecken. Nun gibt es aber fast in jedem Lehrbuch der Maschinensprache die Vorstellung eines solchen Programms, so daß man sich das schönste aussuchen kann. Das Prinzip ist auch da dasselbe, wie wir es von der normalen Division gewohnt sind: Der Divisor wird Schritt für Schritt vom Dividenden abgezogen. In der Literatur [1] fand ich eine sehr kurze Routine, die ich Ihnen leicht modifiziert als Programm 1 vorstellen will.

In Bild 32 ist ein Flußdiagramm dieser Routine gezeigt und in Bild 33 lacht Ihnen wieder das Bit-Gewirr entgegen, das Sie schon von der Multiplikation her kennen, hier aber für die Division.

Damit Sie wissen, wo was hinein- oder herauskommt:

A	: B	= C	+ Rest
↑	↑	↑	↑
\$57/58	\$59/5A	\$57/58	\$5C/5D

An dem folgenden Beispiel soll der Programmverlauf getestet werden: Wir teilen 20867 durch 321. Dabei kommt nach Adam Riese heraus: 65, Rest 2.

In folgender Weise wird in die Speicherzellen die Aufgabe eingespeist:

20867	\$57	1000	0011	LSB
	\$58	0101	0001	MSB
321	\$59	0100	0001	LSB
	\$5A	0000	0001	MSB

Als Ergebnis findet man dann:

65	\$57	0100	0001	LSB
	\$58	0000	0000	MSB
Rest 2	\$5C	0000	0010	LSB
	\$5D	0000	0000	MSB

Als Bit-Zähler dient hier das Y-Register.

b) Erstes Linksschieben des LSB mittels ASL. Dabei gelangt die 1 in das Carry-Bit.

c) Hineinrotieren der 1 aus dem Carry in das MSB mittels ROL.

d), e) Linksrotieren der 16-Bit-Zahl in \$5C/5D, die jetzt noch 0 ist.

f) Situation am Ende der ersten Schleife. Der Bitzähler ist um 1 reduziert.

Im folgenden wird dann jeweils die Situation am Ende der Schleife gezeigt. Beim Berechnen der Differenz muß jeweils darauf geachtet werden, daß die Subtraktion einer Zahl als Addition des Zweierkomplements ausgeführt wird. Das haben wir in Kap. 11 und 14 kennengelernt. Allerdings muß an dieser Stelle nochmal gesagt werden, daß die 1, die zum Einerkomplement hinzuaddiert wird, um das Zweierkomplement zu erhalten, das gesetzte Carry-Bit ist. Nun dürfte es für Sie eigentlich keine Probleme mehr geben, was das Nachvollziehen der Divisionsroutine betrifft.

Damit dürfen wir getrost die 16-Bit-Arithmetik abschließen. Alle vier Grundrechnungsarten können Sie jetzt programmieren. Weitere Rechenarten, wie Potenzieren, das Ziehen von Wurzeln, Logarithmen etc. bedingen ohnehin, daß die Argumente oder Ergebnisse keine Integerzahlen sind. Hier werden wir dann mit Fließkommaarithmetik arbeiten und den dazu vorgesehenen Interpreterroutinen.

```
1 REM ************************************    <250>
2 REM *                                  *    <229>
3 REM *           PROGRAMM  2            *    <125>
4 REM *                                  *    <231>
5 REM *    ERSTELLEN UND AUFRUF EINES    *    <186>
6 REM *         HILFSBILDSCHIRMES        *    <216>
7 REM *                                  *    <234>
8 REM * HEIMO PONNATH      HAMBURG 1985 *    <082>
9 REM ************************************    <002>
10 PRINT CHR$(147):POKE 785,0:POKE 786,96:
   GOTO 30                                    <095>
15 REM --------- UP CURSOR SETZEN ------      <112>
20 POKE 211,SP:POKE 214,Z:SYS 58640:RETURN   <163>
25 REM- ERSTELLEN DES HILFSBILDSCHIRMES-      <123>
30 Z=1:SP=1:GOSUB 20:PRINT"****************
   *************************"                 <151>
40 Z=21:SP=1:GOSUB 20:PRINT"***************
   *************************"                 <211>
50 Z=10:SP=7:GOSUB 20:PRINT"TEST FUER DIE
   VERSCHIEBUNG"                              <110>
55 REM ---- AUFRUF ZUM VERSCHIEBEN ----       <033>
60 A=USR(DUMMY)                               <195>
65 REM ---BILDSCHIRM NEU BESCHREIBEN---       <193>
70 GET A$: IF A$=""THEN 70                    <122>
80 PRINT CHR$(147):Z=2:SP=2:GOSUB 20:PRINT
   "JETZT SOLLTE DER ALTE BILDSCHIRM"         <092>
90 Z=4:SP=2:GOSUB 20:PRINT"UNTER DAS KERNA
   L-ROM GESCHOBEN SEIN"                      <150>
100 PRINT:PRINT:PRINT" -- JEDER{2SPACE}USR
    -AUFRUF HOLT DEN --"                      <003>
110 PRINT" -- HILFSBILDSCHIRM WIEDER .{3SP
    ACE}--"                                   <068>
120 PRINT" -- AUCH IM DIREKT-MODUS{7SPACE}
    --"                                       <056>
130 PRINT:PRINT:PRINT"{2SPACE}PROBIEREN SI
    E MAL: A=USR(1) [RETURN]"                 <050>
140 Z=19:SP=0:GOSUB 20:END                    <164>
```

© 64'er

Programm 2. Das Demo-Programm zur neuen Verschieberoutine. Vorher müssen Programm 3 und Programm 4 geladen werden.

45. Das Programmprojekt wird fortgeführt

Im Kap. 32 haben wir ein Projekt gestartet, das dort eine Kopfzeile rückholbar unter den oberen ROM-Bereich verschob. Unser Wissen ist seither gestiegen und damit auch unsere Ansprüche. Eine Kopfzeile reicht nicht mehr, jetzt soll es ein ganzer Hilfsbildschirm sein, den wir erst in aller Ruhe erstellen wollen, um ihn dann jederzeit abrufbar unter das Betriebssystem zu packen. Den Aufruf wollen wir wieder mit der USR-Funktion steuern. Diesmal soll aber so programmiert werden, daß der Hilfsbildschirm erhalten bleibt und man ihn also mehrfach einblenden kann. Über die Nützlichkeit einer solchen Routine braucht man sicherlich nicht viele Worte zu verlieren: Denken Sie da nur mal an Programme, die irgendwelche Tasten mit besonderen Funktionen belegen, für die Sie eine Gedächtnisstütze brauchen, oder...

Als Programm 2 ist ein kleines Demo-Programm abgedruckt, welches zuerst einen Bildschirm erstellt, dann die Routine »Verschieben« aufruft, den Bildschirm löscht und neu beschreibt und schließlich mit einem weiteren USR den alten Bildschirm einblendet (vorher Programm 3 und 4 laden).

Von nun an können Sie immer – auch im Direktmodus – durch ein USR-Kommando diesen Bildschirm abbilden. Zum Programm in Kap. 32 sind noch zwei Dinge zu bemerken, die hier geändert werden sollen. Erstens eine Frage: Ist Ihnen der Computer mal abgestürzt beim Aufruf des Programms? Die Wahrscheinlichkeit dafür ist ungefähr 1 : 60, wenn nämlich ein Interrupt stattfindet, während die Speicherstelle 1 geändert wird. Obwohl wir erst in den nächsten Kapiteln auf Interrupts eingehen werden, wollen wir die Wahrscheinlichkeit für so einen Absturz auf Null reduzieren. Eine andere Sache ist der Ort, an dem sich das Programm befand. Es hat sich nämlich herausgestellt, daß anscheinend die Nutzung dieses dort gewählten Speicherbereichs nicht ganz so problemlos ist. Bei einigen Anrufen wurde mir erzählt, daß zumindest der Anfang ab $02A7 bei bestimmten Konstellationen überschrieben wird. Deswegen packen wir unser Programm ganz unkonventionell nach $6000, von wo Sie es – das beherrschen Sie ja mit dem SMON inzwischen sicher – dorthin schieben können, wo es Ihnen gefällt. Allerdings müssen dann auch die USR-Adressen geändert werden. Aber auch das dürfte für Sie inzwischen kein Problem mehr sein.

Um diese immerhin schon 2 000 Byte (1 000 für den Bildschirm und nochmal 1 000 für das Farb-RAM) zu verschieben, bedienen wir uns einer Interpreter-Routine, die seit Ausgabe 3/85 des 64'er auch beim Checksummer verwendet wird – der Blockverschiebe-Routine (Bild 34).

Name	**BLTUC**
Zweck	Verschieben von Speicherinhalten im Speicher
Adresse	$A3BF dez. 41919
Vorbereitungen	Quelle Startadresse nach $5F/60 Endadresse+1 nach $5A/5B Ziel Endadresse+1 nach $58/59
Speicherstellen	$58-5B, $5F, $60, $22
Register	Akku, X- und Y-Register
Stapelbedarf	keiner

Bild 34. BLTUC

```
.
,6000  A9 00     LDA #00
,6002  85 5F     STA   5F
,6004  A9 04     LDA #04
,6006  85 60     STA   60
,6008  A9 E8     LDA #E8
,600A  85 5A     STA   5A
,600C  85 58     STA   58
,600E  A9 07     LDA #07
,6010  85 5B     STA   5B
,6012  A9 E3     LDA #E3
,6014  85 59     STA   59
,6016  20 BF A3  JSR A3BF
,6019  A9 00     LDA #00
,601B  85 5F     STA   5F
,601D  A9 D8     LDA #D8
,601F  85 60     STA   60
,6021  A9 E8     LDA #E8
,6023  85 5A     STA   5A
,6025  A9 DB     LDA #DB
,6027  85 5B     STA   5B
,6029  A9 D1     LDA #D1
,602B  85 58     STA   58
,602D  A9 E7     LDA #E7
,602F  85 59     STA   59
,6031  20 BF A3  JSR A3BF
,6034  A9 40     LDA #40
,6036  8D 11 03  STA 0311
,6039  A9 60     LDA #60
,603B  8D 12 03  STA 0312
,603E  60        RTS
------------------------------
.?
```

Programm 3. Erster Teil der Verschieberoutine

```
.
,603F  EA        NOP
,6040  A9 00     LDA #00
,6042  85 5F     STA   5F
,6044  A9 E0     LDA #E0
,6046  85 60     STA   60
,6048  A9 E8     LDA #E8
,604A  85 5A     STA   5A
,604C  85 58     STA   58
,604E  A9 E3     LDA #E3
,6050  85 5B     STA   5B
,6052  A9 07     LDA #07
,6054  85 59     STA   59
,6056  20 77 60  JSR 6077
,6059  A9 E9     LDA #E9
,605B  85 5F     STA   5F
,605D  A9 E3     LDA #E3
,605F  85 60     STA   60
,6061  A9 D1     LDA #D1
,6063  85 5A     STA   5A
,6065  A9 E7     LDA #E7
,6067  85 5B     STA   5B
,6069  A9 E8     LDA #E8
,606B  85 58     STA   58
,606D  A9 DB     LDA #DB
,606F  85 59     STA   59
,6071  20 77 60  JSR 6077
,6074  60        RTS
------------------------------
,6075  EA        NOP
,6076  EA        NOP
,6077  78        SEI
,6078  A5 01     LDA   01
,607A  48        PHA
,607B  A9 35     LDA #35
,607D  85 01     STA   01
,607F  38        SEC
,6080  A5 5A     LDA   5A
,6082  E5 5F     SBC   5F
,6084  85 22     STA   22
,6086  A8        TAY
,6087  A5 5B     LDA   5B
,6089  E5 60     SBC   60
,608B  AA        TAX
,608C  E8        INX
,608D  98        TYA
,608E  F0 23     BEQ 60B3
,6090  A5 5A     LDA   5A
,6092  38        SEC
,6093  E5 22     SBC   22
,6095  85 5A     STA   5A
,6097  B0 03     BCS 609C
,6099  C6 5B     DEC   5B
,609B  38        SEC
,609C  A5 58     LDA   58
,609E  E5 22     SBC   22
,60A0  85 58     STA   58
,60A2  B0 08     BCS 60AC
,60A4  C6 59     DEC   59
,60A6  90 04     BCC 60AC
,60A8  B1 5A     LDA  (5A),Y
,60AA  91 58     STA  (58),Y
,60AC  88        DEY
,60AD  D0 F9     BNE 60A8
,60AF  B1 5A     LDA  (5A),Y
,60B1  91 58     STA  (58),Y
,60B3  C6 5B     DEC   5B
,60B5  C6 59     DEC   59
,60B7  CA        DEX
,60B8  D0 F2     BNE 60AC
,60BA  68        PLA
,60BB  85 01     STA   01
,60BD  58        CLI
,60BE  60        RTS
------------------------------
.?
```

Programm 4. Zweiter Teil der Verschieberoutine

Wieder besteht unser Programm aus zwei Teilen. Im ersten wird der aktuelle Bildschirm nach oben geschoben. Dieser Teil speist lediglich zuerst die Adressen des Bildschirms und des Betriebssystem-ROM in die Abholspeicherstellen der danach aufgerufenen Routine BLTUC und wiederholt diesen Vorgang für die Bildschirmfarbspeicheradressen. Danach verstellen wir noch den USR-Vektor und kehren mit RTS ins Basic-Programm zurück (siehe Programm 3).

Komplexer ist der zweite Teil. Um nämlich die Informationen unter dem ROM lesen zu können, muß dieses ausgeschaltet werden. Leider läßt sich das Betriebssystem-ROM nur zusammen mit dem Basic-Interpreter ausschalten. $A3BF ist aber eine Interpreter-Routine! Da bleibt uns nichts anderes übrig, als diese Routine in unser Programm einzubauen, was uns die Gelegenheit gibt, sie uns mal etwas anzusehen. Als Bild 35 ist sie im Flußdiagramm abgebildet.

Programm 4 zeigt den zweiten Teil unseres Hilfsbildschirm-Programms.

Von $6040 an, wohin wir am Ende des ersten Teils den USR-Vektor gerichtet haben, wird zunächst wieder Quell- und Zielbereich in den Abholspeicherstellen spezifiziert und jeweils danach zuerst für den Bildschirm, dann für das Farb-RAM, das übernommene Unterprogramm angesprungen. Ab $6077 liegt dann das modifizierte Unterprogramm. Die Befehle SEI und CLI gehören zu den wenigen, die Sie erst noch kennenlernen. Sie sind es, die die Absturzwahrscheinlichkeit auf Null bringen. Jedenfalls wird zuerst das ROM aus- und dafür das RAM eingeschaltet. Ab $607F bis $60B9 befindet sich die Interpreter-Routine BLTUC. Darin wird zunächst die Länge des zu verschiebenden Bereichs berechnet, dann festgestellt, ob nur ganze Pages (Seiten) oder auch ein Restbereich verschoben werden soll. Falls ein solcher Restbereich vorhanden ist, wird auch seine Länge berechnet und zuerst dieser verschoben. Daran schließt sich das Ver-

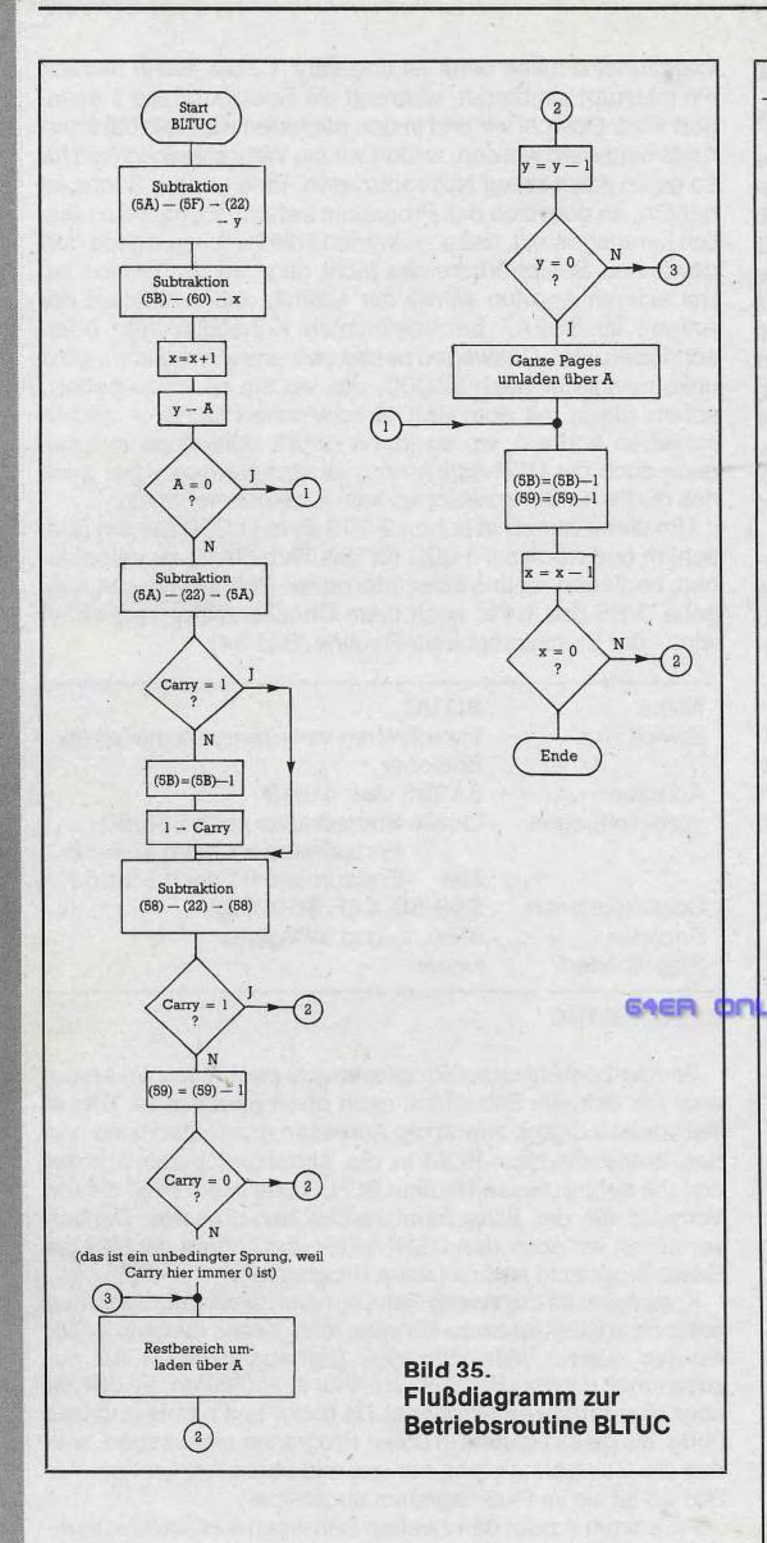


Bild 35. Flußdiagramm zur Betriebsroutine BLTUC

schieben der ganzen Pages an. Das X- und das Y-Register dienen dabei als Zähler.

Ab $60BB schließt sich wieder unsere eigene Routine an, in der wir die ROMs wieder einschalten. Auf diese Weise lassen sich noch mehrere Hilfsbildschirme unter ROM-Bereiche packen. Vielleicht überlegen Sie sich mal dazu einen Weg?

Startadresse	Format	Zahl
$AEA8	MFLPT	Pi
$B1A5	MFLPT	—32768
$B9BC	MFLPT	1
$B9C1	1-Byte-Integer	3
$B9C2	MFLPT	0.434255942
$B9C7	MFLPT	0.576584541
$B9CC	MFLPT	0.961800759
$B9D1	MFLPT	2.88539007
$B9D6	MFLPT	0.707106781 = SQR(1/2)
$B9DB	MFLPT	1.41421356 = SQR(2)
$B9E0	MFLPT	—0.5
$B9E5	MFLPT	0.693147181 = 1n2
$BAF9	MFLPT	10
$BDB3	MFLPT	99999999.9
$BDB8	MFLPT	999999999
$BDBD	MFLPT	1000000000
$BF11	MFLPT	0.5
$BF16	4-Byte-Integer	—100000000
$BF1A	— " —	10000000
$BF1E	— " —	—1000000
$BF22	— " —	100000
$BF26	— " —	—10000
$BF2A	— " —	1000
$BF2E	— " —	—100
$BF32	— " —	10
$BF36	— " —	—1
$BF3A	— " —	—2160000
$BF3E	— " —	216000
$BF42	— " —	—36000
$BF46	— " —	3600
$BF4A	— " —	—600
$BF4E	— " —	60
$BFBF	MFLPT	1.44269504 = 1/1n2
$BFC4	1-Byte-Integer	7
$BFC5	MFLPT	2.14987637E-05
$BFCA	MFLPT	1.43523140E-04
$BFCF	MFLPT	1.34226348E-03
$BFD4	MFLPT	9.61401701E-03
$BFD9	MFLPT	0.0555051269
$BFDE	MFLPT	0.240226385
$BFE3	MFLPT	0.693147186 = 1n2
$BFE8	MFLPT	1
$E08D	MFLPT	11879546
$E092	MFLPT	3.92767774E-08
$E2E0	MFLPT	1.57079633 = Pi/2
$E2E5	MFLPT	6.28318531 = 2*Pi
$E2EA	MFLPT	0.25
$E2EF	1-Byte-Integer	5
$E2F0	MFLPT	—14.3813907
$E2F5	MFLPT	42.0077971
$E2FA	MFLPT	—76.7041703
$E2FF	MFLPT	81.6052237
$E304	MFLPT	—41.3417021
$E309	MFLPT	6.28318531 = 2*Pi
$E33E	1-Byte-Integer	11
$E33F	MFLPT	—6.8473912E—04
$E344	MFLPT	4.85094216E—03
$E349	MFLPT	—0.0161117018
$E34E	MFLPT	0.034209638
$E353	MFLPT	—0.0542791328
$E358	MFLPT	0.0724571965
$E35D	MFLPT	—0.0898023954
$E362	MFLPT	0.110932413
$E367	MFLPT	—0.142839808
$E36C	MFLPT	0.19999912
$E371	MFLPT	—0.333333316
$E376	MFLPT	1
$E3BA	MFLPT	0.811635157
$E8DA - $E8E9	1-Byte-Integers	Tabelle der Farbcodes
$EB81 - $EBC1	— " —	Tastaturdecodierung: Einzelne Tasten
$EBC2 - $EC02	1-Byte-Integers	Tasten mit Shift
$EC03 - $EC43	1-Byte-Integers	Tasten mit Commodore-Taste
$EC78 - $ECB8	1-Byte-Integers	Tasten mit Control-Taste
$ECB9 - $ECE5	1-Byte-Integers	VIC-II-Chip-Registerwerte
$ECF0 - $ED08	1-Byte-Integers	Tabelle der LSBs der Bildschirm-Anfangsadressen

Tabelle 16. Im ROM stehen nicht nur Programme, sondern auch Tabellen, hier einige wichtige Zahlen.

46. Die ROM-Bereiche als Datenquelle

Die ROM-Bereiche enthalten nicht nur ausgeklügelte Maschinenprogramme, sondern auch eine Menge Daten. Sollten Sie mal in die Verlegenheit kommen, beispielsweise die Zahl Pi im MFLPT-Format verwenden zu müssen, dann erfordert das einen ganz schönen Aufwand an Rechen- und Programmarbeit, oder Sie möchten bestimmte Texte wie beispielsweise eine Fehlermeldung verfügbar halten ... und so weiter. Viele von diesen Daten sind schon in der Firmware enthalten und wir werden im folgenden festhalten, wo sie sich befinden und welches Format man vorfindet. Sehen wir uns zunächst Zahlen an (Tabelle 16): Es existieren noch weitere Zahlentabellen in den ROM-Bereichen, die aber selten von Interesse sind. Ebenso wie Zahlen, findet man auch Texte im ROM als ASCII—Werte abgelegt (Tabelle 17)

Sollten Sie mal in die Verlegenheit kommen, solche Texte ausgeben zu wollen, dann legen Sie sie nicht nochmal in einer eigenen Texttabelle ab, sondern schöpfen Sie aus dem Fundus, den wir im ROM-Bereich fix und fertig haben.

Nun noch die Tabelle 18 mit den neuen Assembler-Befehlen.

[1] »Computerspiele und Wissenswertes Commodore 64«, Haar bei München: Markt & Technik Verlag, 1984. Das ist die von P. Lücke besorgte Übersetzung des amerikanischen Buches »More on the sixtyfour« und ist jedem Assembler-Programmierer zu empfehlen.

$A004	CBMBASIC
$A09E - $A19D	Texte der Basic-Befehlsworte (im letzten Byte ist jeweils Bit 7 gesetzt)
$A19E - $A327	Texte der Basic-Fehler- und System-Meldungen. (Im letzten Byte ist jeweils Bit 7 gesetzt)
$A364 - $A38A	Weitere System-Meldungen: OK, ERROR, IN, READY, BREAK. (Das letzte Byte ist jeweils 0)
$ACFC - $AD1D	Fehlermeldungstexte für INPUT: ?EXTRA IGNORED, ?REDO FROM START. (Das letzte Byte ist jeweils 0)
$E460	BASIC BYTES FREE
$E473	**** COMMODORE 64 BASIC V2 **** 64K-RAM-System
$ECE6	LOAD (Return) RUN (Return)
$F0BD - $F12B	Texte für Ein- und Ausgabe-Operationen
$FD10	CBM80

Tabelle 17. Diese Texte sind im ROM als ASCII-Werte abgelegt

Befehls-wort	Adressierung	Byte-zahl	Code Hex	Code Dez	Takt-zyklen	Beeinflussung von Flaggen
LSR	»Akkumulator«	1	1A	26	2	N,Z,C
	absolut	3	4E	78	6	N,Z,C
	0-page-absolut	2	46	70	5	N,Z,C
	absolut-X-indiz.	3	5E	94	7	N,Z,C
	0-page-X-indiz.	2	56	86	6	N,Z,C
ROL	»Akkumulator«	1	2A	42	2	N,Z,C
	absolut	3	2E	46	6	N,Z,C
	0-page-absolut	2	26	38	5	N,Z,C
	absolut-X-indiz.	3	3E	62	7	N,Z,C
	0-page-X-indiz.	2	36	54	6	N,Z,C
ROR	»Akkumulator«	1	6A	106	2	N,Z,C
	absolut	3	6E	110	6	N,Z,C
	0-page-absolut	2	66	102	5	N,Z,C
	absolut-X-indiz.	3	7E	126	7	N,Z,C
	0-page-X-indiz.	2	76	118	6	N,Z,C

Tabelle 18. Die neu besprochenen Assembler-Befehle

47. Was sind Interrupts?

Die Assembler-Befehle haben wir bis auf vier noch offenstehende alle behandelt. Diese vier, die alle mit dem Interrupt-Handling zusammenhängen, sollen nun unser Thema sein. Wenn wir sie beherrschen, haben wir den ersten Schritt zum Meister der Assembler-Alchimie getan. Diese vier kleinen 1-Byte-Befehle öffnen uns eine geheime Pforte zu einem Universum an Programmier-Möglichkeiten, von dem wir bisher kaum zu träumen vermochten. Genug der Schwärmerei, erst kommt noch eine Menge Arbeit, die uns wohl mehrere Kapitel in Atem halten wird.

Zuvor noch eine Bemerkung: es gibt kaum ein Thema im Rahmen der Programmierung in Assembler, welches so penetrant häufig Abstürze provoziert, wie das nunmehr angesteuerte! Falls Sie noch keine RESET-Taste an Ihrem Computer haben, wird es nun höchste Zeit. Diese nützlichen Dinger werden inzwischen schon so preiswert angeboten (sehen Sie mal in den Kleinanzeigenteil!), daß Sie zur Grundausstattung eines Assembler Alchimisten zählen.

Unser Computer ist – solange er eingeschaltet ist – ständig mit irgendwelchen Tätigkeiten beschäftigt. Im Direktmodus hängt er beispielsweise meistens in einer Warteschleife und harrt der Eingaben, im Programm-Modus arbeitet er sich mit Hilfe der Interpreterschleife durch einen Basic-Befehlstext hindurch und so weiter. Nun werden Sie ja sicher schon festgestellt haben, daß er im Direktmodus auch den Cursor blinken läßt, in beiden Modi die TI$-Uhr weiterzählt und weitere Dinge macht, die anscheinend so nebenher passieren. Schon in Kapitel 8 aber haben wir einen Unterschied zwischen Mensch und Computer festgehalten: Der Mensch kann mehrere Dinge gleichzeitig tun, der Mikroprozessor ist nur fähig zu einer Arbeit pro Zeiteinheit. Weil aber diese Zeiteinheiten so unfaßbar kurz sind (etwa eine Millionstel Sekunde), haben wir Benutzer den Eindruck der Gleichzeitigkeit.

Wenn dem aber so ist, wie macht es der Computer, daß er beispielsweise ein Programm abarbeitet und trotzdem die TI$-Uhr weiterzählt? Durch Unterbrechungen (interrupt = unterbrechen) der gerade ausgeübten Tätigkeit. Ein Beispiel aus dem täglichen Leben soll uns das illustrieren: Sie lesen gerade diesen Artikel, da klingelt das Telefon und ein Freund möchte von Ihnen wissen, was eigentlich Unterbrechungen sind. Während Sie es ihm erklären, fängt in der Küche der Teekessel schrill zu pfeifen an. Sie sagen Ihrem Freund, er möge sich einen Moment gedulden, gehen in die Küche und nehmen den Kessel vom Feuer. Dann kehren Sie ans Telefon zurück und beenden nach einer Weile das Gespräch. Nach dem Auflegen des Telefonhörers setzen Sie die Lektüre des

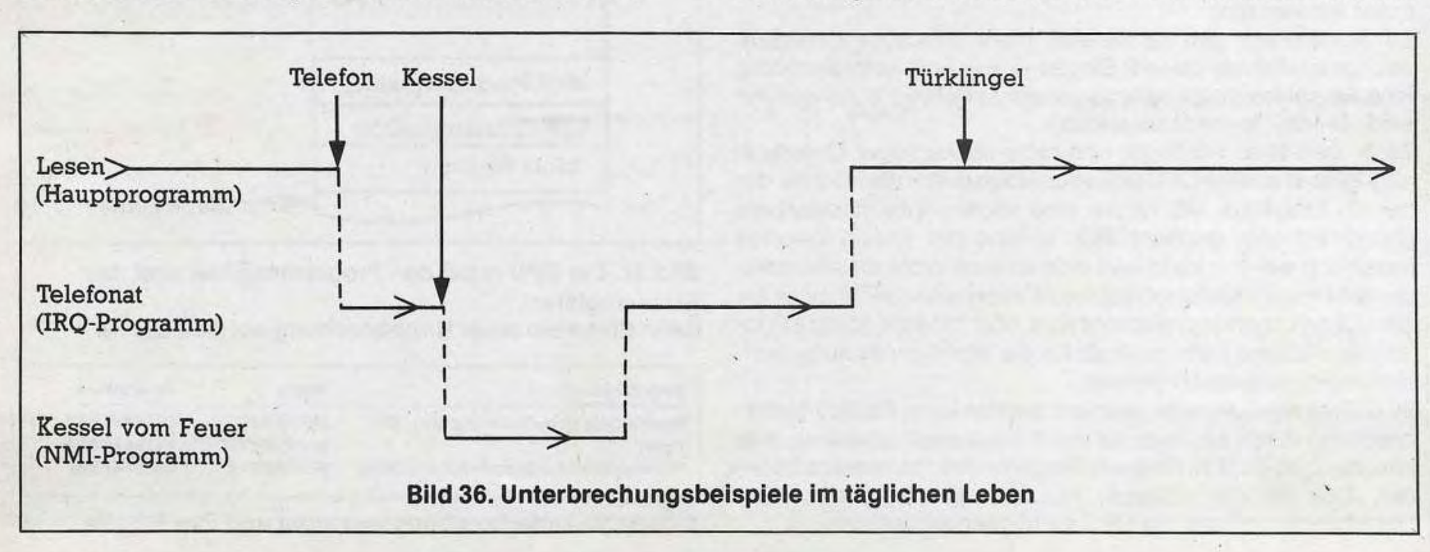


Bild 36. Unterbrechungsbeispiele im täglichen Leben

Artikels fort, fest entschlossen, sich nun nicht mehr unterbrechen zu lassen. Kurze Zeit später klingelt jemand an der Tür. Sie lassen sich dadurch nicht stören.

Dieses Gleichnis gibt ziemlich genau wieder, was sich im Computer - nur bei millionenfacher Geschwindigkeit - bei Unterbrechungen abspielt. In Bild 36 ist das Schema des Ablaufes grafisch dargestellt. In gewisser Weise ähnelt das ganze dem Abarbeiten von Unterprogrammsequenzen. Weshalb programmiert man dann nicht einfach mittels einiger JSR-Aufrufe? Dafür hat L.A.Leventhal einen einleuchtenden Vergleich: »Ein Unterbrechungs-System entspricht etwa einer Telefonklingel. Sie läutet, wenn ein Anruf empfangen wird, so daß man den Hörer nicht laufend abnehmen muß, um festzustellen, ob sich jemand in der Leitung befindet.« (L.A.Leventhal, »6502 Programmieren in Assembler«, München te-wi Verlag, Seite 12-1). Unterbrechungen können dann angefordert und abgearbeitet werden, wenn sie nötig sind, im Gegensatz zu Unterprogrammen, die erst dann berücksichtigt werden, wenn der Programmzähler einen JSR-Befehl erfaßt. Um also schnell reagieren zu können, müßte man sehr oft in einem Programm eine Unterroutine anspringen, die auf gewisse Registerinhalte prüft und dann zur Bearbeitung verzweigt oder - bei Nichtvorliegen einer Bedingung - im normalen Programm weiterfährt. Das kostet unnötig Zeit und Speicherraum. Mancher Verkehr des Computers mit Peripherie erfordert so schnelle Reaktionen, daß diese nur geleistet werden können durch Unterbrechen des laufenden Programmes.

Ich denke, daß Sie nun die Notwendigkeit von Unterbrechungen erkennen. Fast jede CPU kennt solche Unterbrechungssysteme. Man kann sie charakterisieren durch die Beantwortung folgender Fragen:

1) Welche Unterbrechungs-Eingänge weist die CPU auf?
2) Wie reagiert die CPU auf eine Unterbrechung?
3) Wie bestimmt die CPU die Unterbrechungsquelle, wenn die Anzahl der Quellen größer ist als die Anzahl der Eingänge?
4) Kann die CPU zwischen wichtigen und weniger wichtigen Unterbrechungen unterscheiden?
5) Wie und wann wird das Unterbrechungssystem freigegeben oder gesperrt?

All diese Fragen werden wir für unseren Computer ergründen.

48. Das Unterbrechungssystem der CPU 6510/6502

Einige dieser Charakteristika sind schnell zu zeigen:
Zu 1: Unsere CPU hat genau 2 Eingänge für Unterbrechungen (wenn man RESET außer acht läßt, was wir im folgenden meist tun werden).
Zu 3: Natürlich gibt es weitaus mehr denkbare Unterbrechungsquellen als diese 2 Eingänge, weshalb softwaremäßig eine Registerabfrage (das sogenannte Polling) durchgeführt wird, um die Quelle festzustellen.
Zu 4: Zwischen wichtiger und nicht so wichtiger Unterbrechung kann unsere CPU unterscheiden durch die Priorität der beiden Eingänge. Wir haben eine sogenannte maskierbare Unterbrechung, genannt IRQ, welche per Befehl ignoriert (maskiert) werden kann und eine andere, nicht maskierbare, die daher auch NMI (not maskable interrupt = nicht maskierbare Unterbrechung) genannt wird. NMI hat eine höhere Priorität als IRQ und kann deshalb für die wichtigeren Aufgabenstellungen eingesetzt werden.
Zu 5: Freigegeben oder gesperrt werden kann die IRQ-Unterbrechung durch ein Sperrbit (auch Maskenbit genannt), welches sich als Bit 2 im Flaggen-Register des Prozessors befindet. Das ist die I-Flagge. Für den Empfang der NMI-Unterbrechung kann die CPU nicht gesperrt werden.

Um mal die Parallele zu unserem Beispiel zu zeigen: Das Lesen des Artikels ist die gerade stattfindende Tätigkeit des Computers. Die Telefonklingel signalisiert einen IRQ, der im folgenden bearbeitet wird. Das Pfeifen des Teekessels soll einem NMI entsprechen. Wenn dieser dann bearbeitet ist, geht es mit der Abarbeitung des IRQ weiter. Nach Beendigung des Telefonates wird das Unterbrechungs-Sperrbit gesetzt (sie nehmen sich vor, sich nicht mehr stören zu lassen) und mit der normalen Tätigkeit fortgefahren. Weil der nun folgende IRQ damit maskiert ist, wird das Türklingeln ignoriert.

Die Frage 2, nämlich wie unsere CPU auf eine Unterbrechung reagiert, blieb noch unbeantwortet. Nun soll sie behandelt werden:
a) Am Ende jedes Befehls überprüft die CPU automatisch den Zustand des Unterbrechungs-Systems. Wenn an einer der beiden Unterbrechungsleitungen eine Anforderung vorliegt und diese auch freigegeben ist, beginnt die Unterbrechung zu wirken.
b) Zunächst wird der Programmzählerinhalt in der Reihenfolge MSB, LSB auf den Stapel geschrieben. Danach wandert noch der Prozessorstatus auf den Stapel (siehe Bild 37).
c) Durch Setzen des Unterbrechungs-Sperrbits I werden weitere maskierbare Unterbrechungen (IRQ) unterbunden.
d) Nun holt sich die CPU aus einem Vektor ganz am Ende des Speichers eine Adresse, lädt diese in den Programmzähler und startet auf diese Weise ein Serviceprogramm, das dem auslösenden Anlaß Rechnung trägt. In der Tabelle 19 sind die zu den Unterbrechungsformen und zum RESET gehörigen Vektoren aufgeführt.

Bevor wir uns weiter mit den so angesteuerten Routinen befassen, wollen wir die 4 Befehle kennenlernen, die uns noch fehlen.

49. Schlüssel zur Unterbrechungsprogrammierung: CLI, SEI, RTI, BRK

Das Sperren der maskierbaren Unterbrechung IRQ und das Löschen der Maske erfolgt durch Setzen oder Löschen des Sperrbits im Prozessorstatus-Register. Dieses Bit, die I-Flagge, kann durch den Befehl CLI gelöscht werden. **CLI**

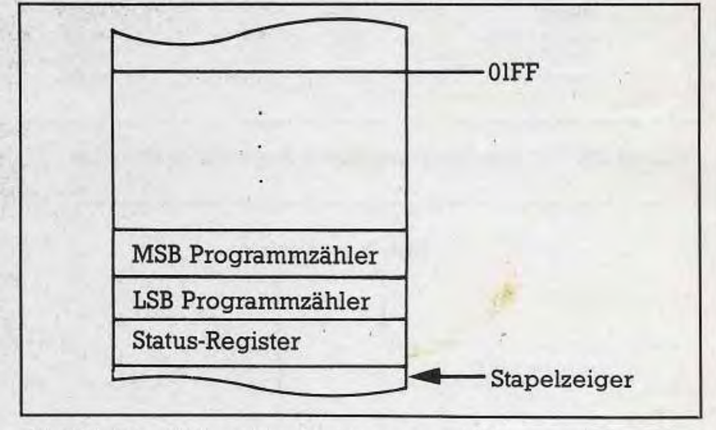


Bild 37. Die CPU rettet den Programmzähler und das Statusregister beim Eintreten einer Unterbrechung auf den Stapel

Unterbrechungsart	Vektor	Zieladresse
Maskierbare Unterbrechung (IRQ, BRK)	$FFFE/FFFF	65352 $FF48
Reset	$FFFC/FFFD	64738 $FCE2
Nichtmaskierbare Unterbrechung (NMI)	$FFFA/FFFB	65091 $FE43

Tabelle 19. Unterbrechungsvektoren und ihre Inhalte

kommt von »CLear Interrupt mask«, was bedeutet »lösche die Unterbrechungs-Maske«. Immer dann, wenn IRQs zugelassen sein sollen zur Bearbeitung durch den Mikroprozessor, muß damit die I-Flagge gelöscht werden. Wie Sie sehen, ist CLI ein 1-Byte-Befehl mit impliziter Adressierung. Er braucht genau 2 Taktzyklen zur Erledigung seiner Aufgabe.

Wenn wir später eigene Unterbrechungsroutinen schreiben, stehen wir oft vor der Frage, ob wir innerhalb unseres Unterbrechungsprogramms weitere Unterbrechungen zulassen wollen. Manchmal ist das wichtig, beispielsweise bei der Tastaturabfrage. Wie wir vorhin erwähnt haben, sperrt die CPU automatisch bei der Annahme von Unterbrechungen weitere IRQs durch Setzen der I-Flagge. Einer der ersten Befehle der eigenen Unterbrechungsroutine wird dann die Freigabe von Unterbrechungen sein durch Löschen der I-Flagge.

SEI bewirkt das Gegenteil von CLI. Der Befehl setzt die I-Flagge auf 1 (»**SE**t **I**nterrupt mask«) und verhindert, daß der Mikroprozessor weiteren IRQs seine Aufmerksamkeit schenkt. Das ist in den Fällen wichtig, in denen beispielsweise störungsfrei der Inhalt des Charakter-ROM gelesen werden soll oder während der Änderung von Speicherstellen, die die IRQ-Routine benutzt. Wie wichtig das Sperren von IRQs sein kann, haben Sie eventuell bemerkt, wenn Ihnen das Hilfsbildschirmprogramm aus Kap. 32 mal abgestürzt ist. Seit der letzten Folge – wo wir die IRQs gesperrt haben – ist Ihnen das sicherlich nicht mehr passiert. Ebenso wie CLI ist SEI ein 1-Byte-Befehl mit impliziter Adressierung, und auch er braucht 2 Taktzyklen zur Bearbeitung.

Noch eine Bemerkung zum Verhindern der IRQs. Wir werden später sehen, was alles während der 60mal pro Sekunde aufgerufenen Unterbrechung erledigt wird. Jede Routine, die SEI verwendet, verbraucht Rechenzeit. Wenn sie so lange dauert, daß eine oder mehrere dieser regelmäßigen IRQs unterbunden werden, kann das unter Umständen zu Störungen von Programmabläufen führen. In solchen Fällen ist es sinnvoll, in die eigene Routine den Teil der regulären IRQ-Routine einzubauen, der im Programmablauf durch sein Fehlen Störungen verursacht. Meistens kann man aber durch gute Planung eines Programmes dieses Problem umgehen.

RTI heißt »**R**e**T**urn from **I**nterrupt«, zu deutsch also: »kehre aus dem Unterbrechungsprogramm zurück.« Es entspricht in seinem Einsatz etwa dem RTS bei Unterprogrammrücksprüngen. Während RTS aber lediglich den alten Programmzählerinhalt vom Stapel holt (und noch eine 1 dazuaddiert), schafft RTI auch noch den alten Inhalt des Status-Registers vom Stapel zurück. Der genaue Ablauf ist wie folgt:

1) Alten Prozessorstatus vom Stapel wieder ins Status-Register schieben.
2) Stapelzeiger um 1 erhöhen
3) LSB des alten Programmzählers vom Stapel nehmen und zurückschreiben.
4) Stapelzeiger um 1 erhöhen
5) MSB des alten Programmzählers vom Stapel nehmen und zurückschreiben.
6) Stapelzeiger um 1 erhöhen.

Damit ist der Zustand vor der Unterbrechung wiederhergestellt. Auch die I-Flagge ist so automatisch wieder gelöscht, denn vor der Unterbrechung war sie sicher nicht gesetzt gewesen und der alte Status-Zustand ist ja jetzt wieder vorhanden.

RTI ist ebenfalls ein 1-Byte-Befehl mit impliziter Adressierung. Seine vollständige Bearbeitung dauert 6 Taktzyklen.

Bei eigenen Unterbrechungs-Routinen verwendet man häufig nicht RTI, sondern springt durch JMP an eine sinnvolle Stelle des normalen Unterbrechungsprogrammes. Auf diese Weise kann man dann die normalen Arbeitsgänge der vorprogrammierten Unterbrechung oder Teile davon noch ausführen lassen.

Den Befehl **BRK** (break=Software-Unterbrechung) haben wir schon verwendet. Er entspricht in seinem Einsatz etwa dem STOP-Befehl in Basic und dient wie jener Befehl dort hauptsächlich dem Testen von Programmen. Tatsächlich unterscheidet sich die Reaktion unserer CPU bei Auftreten eines BRK kaum von der bei einem IRQ. Folgendes passiert:

a) Der Programmzähler wird um 2 erhöht.
b) Bit 4 des Prozessorstatusregisters, die Break-Flagge B, wird auf 1 gesetzt.
c) Das MSB des Programmzählers wird auf den Stapel gebracht und der Stapelzähler um 1 heruntergezählt.
d) Dasselbe geschieht nun mit dem LSB des Programmzählers
e) und mit dem Statusregister.
f) Das Unterbrechungsmaskenbit, die I-Flagge, wird auf 1 gesetzt um IRQs zu sperren.
g) In den Programmzähler wird nun aus dem Vektor FFFE/FFFF dieselbe Adresse geladen, die auch bei IRQs benutzt wird. Damit startet nun das Programm, das diese Unterbrechung bearbeitet.

Sie sehen, daß der BRK-Befehl ein ziemlich komplizierter Geselle ist. Zwar handelt es sich wieder um einen 1-Byte-Befehl mit impliziter Adressierung, aber er benötigt immerhin 7 Taktzyklen, um all diese Arbeit zu bewältigen.

Wir haben BRK bisher immer zur Programmunterbrechung mit nachfolgender Registeranzeige durch den SMON eingesetzt. Der SMON ist – wie fast jeder Monitor – so programmiert, daß ein BRK zur Registeranzeige führt. Das ist natürlich sinnvoll beim Einsatz von BRK zur Fehlersuche. In dem Moment, wo ein BRK vom Prozessor bearbeitet wurde, kann nur durch die gesetzte B-Flagge von einem IRQ unterschieden werden. Es ist manchmal nötig, schon zu diesem Zeitpunkt diesen Unterschied festzustellen. Deshalb verwendet man den nachfolgend beschriebenen Test zu diesem Zweck:

PLA

in den Akku wird das zuletzt auf den Stapel geschobene Prozessorstatus-Register geholt.

PHA

und sogleich wieder zurückgeschoben

AND #$10

durch die AND-Verknüpfung mit der Binärzahl 0001 0000 kann eine eventuell vorhandene B-Flagge isoliert werden.

BNE BREAK

Falls eine B-Flagge gesetzt war, ist der Akku ungleich 0 und die Bearbeitung verzweigt zum von uns konstruierten BREAK-Programm. War der Akku nach dieser AND-Verknüpfung gleich 0, dann erfolgt keine Verzweigung und es handelt sich um einen IRQ, zu dessen Bearbeitung nun zu springen ist.

Es gibt noch eine andere – gebräuchlichere – Möglichkeit, zwischen einem BRK und einem IRQ zu unterscheiden, die allerdings erst zu einem späteren Zeitpunkt des computerinternen Unterprogrammes erfolgt. Von dieser zweiten Möglichkeit wird im SMON Gebrauch gemacht und wir werden sie nachher auch kennenlernen.

Natürlich kann der BRK-Befehl auch zu anderen Zwecken als zur Registeranzeige durch einen Monitor verwendet werden. Es kommt immer darauf an, welches Service-Programm wir dem Computer anbieten. Springt man aus so einem Service-Programm mittels RTI zurück ins Hauptprogramm, dann muß man berücksichtigen, daß der Programmzähler vor der Sicherung auf dem Stapel um 2 erhöht worden ist. Manchmal sind deshalb noch Korrekturen des Programms nötig.

Ich hoffe, daß Sie bisher diesen Artikel nicht zu frustrierend fanden, denn ständig ist die Rede vom eigenen Unterbrechungs-Programm und dabei wissen Sie – außer durch BRK – noch gar keine Möglichkeit, einen IRQ oder NMI auszulösen, und Sie sind sicher noch sehr vorsichtig mit dem Gedanken an eigene Unterbrechungs-Routinen, weil Ihnen ja noch

unbekannt ist, wie die normale Firmware Unterbrechungen behandelt. Keine Angst: All das werden wir noch klären. Betrachten Sie diesen Teil zum Thema Unterbrechungen vielleicht mehr wie ein Handbuch, in dem Sie dann, wenn Ihr Verständnis gestiegen ist, nochmal zurückblättern können.

Wir haben bisher nur betrachtet, wie unsere CPU reagiert, wenn an einem der beiden Unterbrechungs-Eingänge (IRQ und NMI) eine Unterbrechungs-Anforderung vorliegt. Um nun aber selbst ins Geschehen eingreifen zu können, ist es nötig zu wissen, wie diese Anforderung dorthin gelangt. Das erfordert von uns die Beschäftigung mit anderen Computerbausteinen als der CPU, die bisher im Mittelpunkt unseres Interesses stand.

50. Woher kommen die Unterbrechungs-Anforderungen?

Quellen für Unterbrechungen können viele genannt werden: Diskettenstation, Datasette, Drucker, Modem, Schaltelemente und so weiter. Um aber eine gewisse Übersicht zu bekommen, sollte man unterscheiden zwischen primären und sekundären Unterbrechungsquellen. Das soll kurz erläutert werden: Die Diskettenstation beispielsweise ist über den seriellen Port mit dem Computer verbunden. Dieser wiederum steht in direktem Kontakt zu 2 Bausteinen, den CIAs. Erst diese CIAs stehen in direktem Kontakt zur CPU. Alle Unterbrechungs-Quellen, die direkt Signale an die beiden Unterbrechungseingänge unserer CPU senden, sollen künftig »primäre« Quellen genannt werden, die anderen, die nur über solch eine primäre Quelle Unterbrechungs-Anforderungen stellen, werden von uns als »sekundäre« Quellen bezeichnet. Weil wir irgendwo einen Schnitt machen müssen - einmal, um nicht völlig auszuufern in der Erklärung von peripheren Geräten (das soll anderen, kompetenteren überlassen bleiben) und zum anderen, weil ich mich da auch nicht so gut auskenne - werden wir uns im folgenden auf die primären Unterbrechungsquellen beschränken. Da bleibt aber noch mehr als genug zu tun übrig und deshalb soll auch nur eine Auswahl dieser Primärquellen detailliert behandelt werden.

Welches sind nun die primären Unterbrechungsquellen? Hier sind sie aufgeführt:

1) Der VIC-II-Chip (MOS 6566/6567 Video Interface Controller)
2) Die beiden CIAs (MOS 6526 Complex Interface Adapter)
3) Die RESTORE-Taste
4) Der Expansion-Port
5) RESET (paßt hier nicht ganz her, woanders aber auch nicht besser)

Den Expansion-Port werden wir nicht behandeln und einen RESET nur ziemlich kurz betrachten, weil es sich dabei eigentlich nicht um eine Unterbrechung im bisher definierten Sinn handelt.

51. Der VIC-II-Chip als Unterbrechungsquelle

Soweit ich feststellen konnte, kommt der VIC-II-Chip in Bezug auf unsere CPU nur als Anforderer von maskierten Unterbrechungen (IRQ) in Frage. Die Handhabung seiner Unterbrechungs-Verlangen geschieht im VIC-II-Chip durch zwei Register. Vier Ereignisse sind eingeplant, deren Eintreten zur Unterbrechung führen kann:
1) Rasterzeilen-Unterbrechung
2) Kollision eines Sprites mit Hintergrund
3) Kollision von Sprites untereinander
4) Lichtgriffel-Unterbrechung.

Die ersten 3 Auslöser werden wir uns in kommenden Folgen genau ansehen und dabei vielerlei interessante Möglichkeiten feststellen. Die Option, die der Lichtgriffel bietet, wird nicht behandelt werden: Meine Kenntnisse auf diesem Sektor sind nur gering (nobody is perfect).

Das sogenannte Interrupt Enable Register (Unterbrechungs-Zulassungs-Register) des VIC-II-Chips ist Register 26. Es befindet sich in der Speicherstelle 53274 ($D01A) (siehe Bild 38).

In diesem Register wird festgelegt, ob eines - oder mehrere - der 4 möglichen auslösenden Ereignisse eine Unterbrechungsanforderung an den Mikroprozessor senden soll. Jedem Ereignis ist ein Bit zugeordnet. Ist dieses Bit gleich 1, dann ist die Unterbrechung freigegeben, ist es gleich 0, dann liegt eine Sperrung vor. Die Zuordnung der Bits ist wie folgt:

Bit 0	Rasterzeilen-IRQ
Bit 1	Sprite/Hintergrund-Kollision
Bit 2	Sprite/Sprite-Kollision
Bit 3	Lichtgriffel-IRQ
Bits 4 bis 7	sind ungenutzt und haben immer den Wert 1.

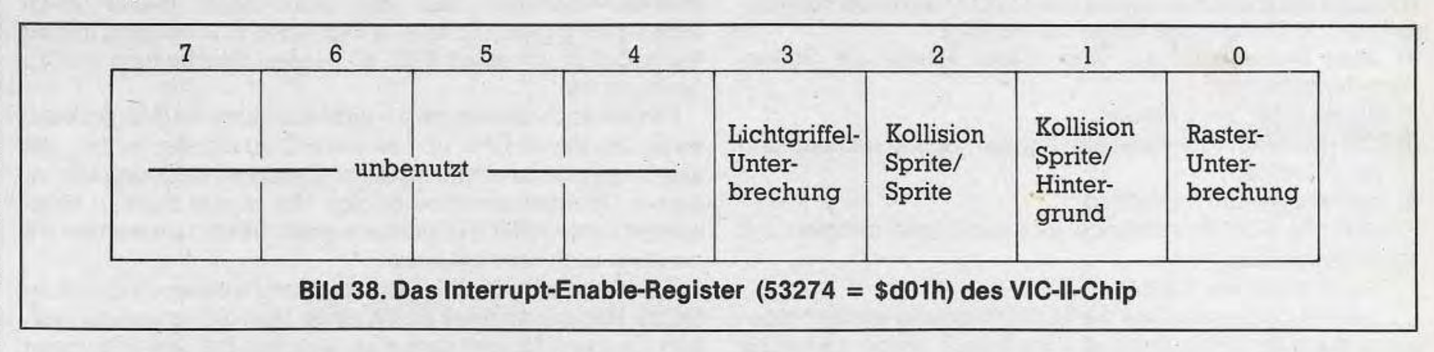


Bild 38. Das Interrupt-Enable-Register (53274 = $d01h) des VIC-II-Chip

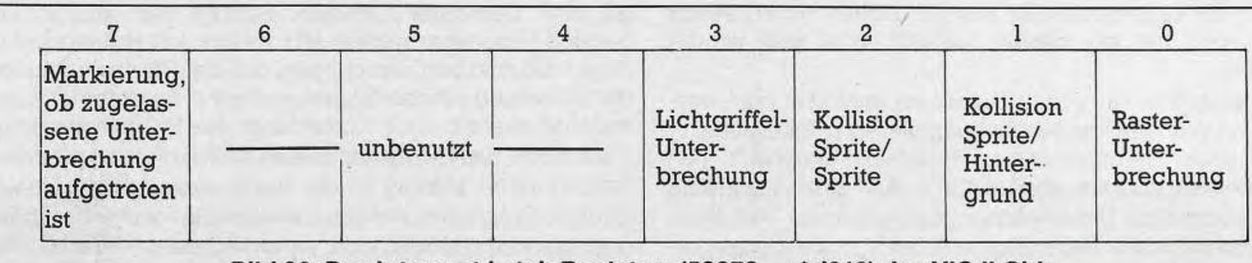


Bild 39. Das Interrupt-Latch-Register (53273 = $d019) des VIC-II-Chip

Das Register 25 wird Interrupt Latch Register genannt, was etwa zu übersetzen wäre mit »Unterbrechungs-Einrast-Register« (siehe Bild 39). Der englische Ausdruck »latch«, der nur umschreibend oder sehr technisch übersetzt werden kann, beschreibt eigentlich recht genau, was in diesem Register geschieht. Ein »latch« ist nämlich so etwas wie ein Schnappriegel, also ein Riegel, der bei der Betätigung einrastet. Wenn eines der 4 möglichen Ereignisse eintritt, schnappt im dazugehörigen Bit dieses Registers der Inhalt auf 1. Die Bit-Zuordnung ist die gleiche wie in Register 26. Aber das Bit 7 hat hier noch eine Bedeutung: Ist eines der Bits 0 bis 3 auf 1 gesetzt und das dazugehörige Ereignis in Register 26 auch zur Unterbrechung zugelassen (also auch dort gleich 1), dann taucht in Register 25, Bit 7 eine 1 auf. So kann durch einfaches Lesen dieses Bits festgestellt werden, ob ein IRQ durch den VIC-II-Chip ausgelöst wurde.

Will man in diesem Register ein gesetztes Bit löschen, muß man – außergewöhnlich! – eine 1 in die Bitposition schreiben.

Mit Recht erwarten Sie nun eigentlich eine Anwendung des bisher Gelernten. Bei Unterbrechungsprogrammen ist es aber dringend nötig, immer den gesamten Komplex im Auge zu haben. Ich habe mich daher entschlossen, zuerst alles zu erklären und dann Anwendungsmöglichkeiten vorzustellen. Ihre Geduld wird auf eine harte Probe gestellt, aber ich hoffe, daß Sie später feststellen, daß es sich gelohnt hat, etwas zu warten.

52. Die beiden CIA-Bausteine als Unterbrechungsquellen

An sich sind die beiden CIAs in unserem Computer völlig identisch. Sie werden aber unterschiedlich eingesetzt. Sehen wir uns zunächst einmal an, was beiden in Bezug auf Unterbrechungen gemeinsam ist, um danach die Unterschiede festzuhalten. Die Unterbrechungs-Steuerung geschieht in Register 13 dieser Bausteine. Dieses Register hat 2 Funktionen: Es bestimmt, ob eine Unterbrechungsanforderung an die CPU gesandt werden soll, und es stellt fest, ob ein Ereignis stattgefunden hat, das zur Unterbrechung führen kann. Die Bedienung dieses Registers ist demzufolge auch etwas unübersichtlich, aber wir haben schon ganz andere Probleme gemeistert.

Sehen wir uns zuerst einmal an, welche Ereignisse vom Standpunkt eines CIA-Bausteines als Unterbrechungskriterium dienen können:

1) Unterlauf der Uhr A
2) Unterlauf der Uhr B
3) Die interne Uhr hat eine Alarmzeit erreicht
4) Am SP-Eingang (hängt mit dem seriellen Port zusammen) ist ein bestimmter Zustand erreicht
5) An einem Eingang namens FLAG ist ein bestimmter Zustand erreicht.

Die Ereignisse 4 und 5 werden wir ebenfalls im weiteren weitgehend ausklammern.

Nun zum Register 13, dem Unterbrechungs-Kontroll-Register (siehe Bild 40).

Auch hier gehört zu jedem Ereignis ein Bit. Dabei – um Wiederholungen zu vermeiden – ist die Zuordnung schon durch die eben angegebene Ereignisaufzählung gegeben. Ziehen Sie von der vorangestellten Nummer immer eine 1 ab und Sie haben die Bitnummer. Die Bits 5 und 6 sind unbenutzt. Bit 7 hat eine dreifache Funktion, die eng mit den anderen Bitinhalten verknüpft ist. Sehen wir uns das mal der Reihe nach an:

Lesen des Registers

Sind Unterbrechungsereignisse aufgetreten, dann sind die dazugehörigen Bits auf 1 gesetzt. Bit 7 ist gleich 1, wenn mindestens ein solches Ereignis stattgefunden hat und außerdem dieses Ereignis als Unterbrechungsauslöser freigegeben ist. Auf diese Weise kann – ähnlich wie beim VIC-II-Chip-Register 25 – festgestellt werden, ob die Unterbrechung durch einen der beiden CIAs angefordert wurde. Im Unterschied aber zum VIC-II-Register wird Register 13 durch das Lesen gelöscht. Braucht man den Inhalt also noch, sollte man ihn irgendwo zwischenspeichern.

Schreiben in das Register

Bit 7 = 0 erzeugt Sperren.

Das erkennt man am besten an einem Beispiel. Nehmen wir an, wir möchten die Unterbrechung sperren, die durch einen Unterlauf von Uhr A erzeugt werden kann. Das betrifft das Bit 0. Wir schreiben in das Register 13 folgende Zahl: 0000 0001

Wie Sie sehen, ist das Bit 7 gleich 0. Die 1 in Bit 1 bewirkt die Sperrung. Durch die Nullen in den anderen Bits wird bewirkt, daß die anderen Unterbrechungs-Ereignisse nicht beeinflußt werden. Wollten wir alle sperren, dann müßten wir einschreiben: 0001 1111

Auf diese Weise können selektiv einzelne Unterbrechungen durch Einschreiben der 1 bei gelöschtem Bit 7 gesperrt werden.

Bit 7 = [illegible] erzeugt Freigabe.

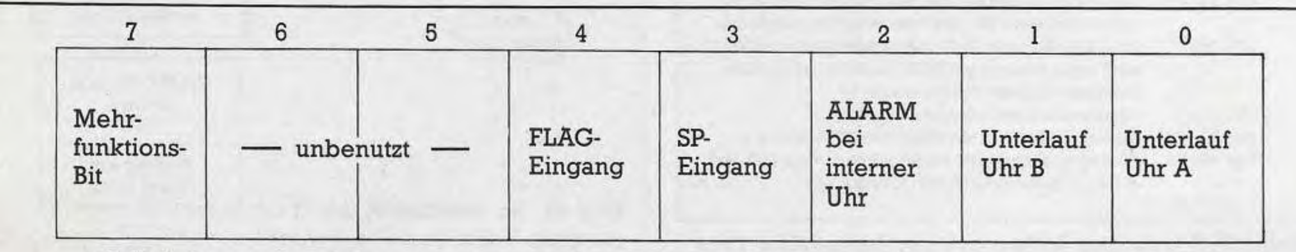
Auch hier wieder ein Beispiel. Wenn wir ganz gezielt Unterbrechungen durch Unterlauf der Uhr A freigeben wollen, müssen wir die folgende Zahl in Register 13 schreiben: 1000 0001

Bit 7 (gleich 1) zeigt an, daß diejenigen Unterbrechungen freizugeben sind, deren Bits auf 1 gesetzt sind. Alle anderen Unterbrechungen, wo also in der dazugehörigen Bitposition der einzuschreibenden Zahl eine 0 steht, bleiben unverändert.

Ein wichtiger Unterschied zwischen den beiden CIAs ist der, daß der Unterbrechungsausgang von CIA 1 mit dem IRQ-Eingang der CPU verbunden ist, wohingegen der entsprechende Ausgang von CIA2 an den NMI-Eingang unseres Mikroprozessors führt. Daher löst der CIA 1 nur IRQs aus, er wird manchmal deshalb auch IRQ-CIA genannt. Der andere ist dann der NMI-CIA, weil er nur NMIs anfordern kann.

53. Der IRQ-CIA

Das Register 13 des IRQ-CIA (der die Speicherstellen 56320 bis 56335 belegt), liegt in Zelle 56333 ($DC0D). Die einzel-



7	6	5	4	3	2	1	0
Mehr-funktions-Bit	— unbenutzt —		FLAG-Eingang	SP-Eingang	ALARM bei interner Uhr	Unterlauf Uhr B	Unterlauf Uhr A

Bild 40. Genereller Aufbau der Unterbrechungs-Kontroll-Register (13) der beiden CIA-Bausteine

nen Bits sind wie folgt zugeordnet:
Bit 0 Unterlauf Uhr A
Von hier kommt der IRQ, der 60mal pro Sekunde stattfindet zur Tastaturabfrage, zum Weiterstellen der TI$-Uhr etc.
Bit 1 Unterlauf Uhr B
Spielt bei Kassettenoperationen und dem seriellen Port eine Rolle.
Bit 2 ALARM bei interner Uhr.
Spielt beim Zufallszahlengenerator (RND(0)) eine Rolle.
Bit 3 Hier kommen durch den User-Port Unterbrechungs-Anforderungen.
Bit 4 ist verbunden mit dem seriellen Port und der Kassetten-Lese-Leitung.

54. Der NMI-CIA

Ebenso kurz und schmerzlos wie beim CIA 1 soll auch das besondere am CIA 2, dem NMI-CIA (er belegt den Speicher von 56576 bis 56831) vorgestellt werden. Sein Register 13 findet sich in Speicherstelle 56589 ($DD0D). Die Bits 0 und 1 (Unterläufe der beiden Uhren) spielen beim Senden beziehungsweise Empfangen von Daten über die RS232C-Schnittstelle eine Rolle, Bit 2 (ALARM) wird nicht verwendet, Bit 3 ist direkt mit dem User-Port verbunden ebenso wie Bit 4. Der NMI-CIA wird uns in seiner normalen Funktion nicht mehr beschäftigen.

55. Die RESTORE-Taste und ein kleines Testprogramm

Die RESTORE-Taste ist direkt mit dem NMI-Eingang unseres Mikroprozessors verbunden. Das ermöglicht es uns, durch einfaches Drücken dieser Taste jederzeit ins Geschehen einzugreifen, ohne uns um Details kümmern zu müssen, ob sich der Computer gerade im Direkt- oder im Programm-Modus befindet und so weiter. Denn NMI hat die höchste Priorität der Unterbrechungen.

Ein kleines Testprogramm soll Ihnen hier noch vorgestellt werden, das Sie vielleicht aber noch nicht ganz verstehen werden, weil wir erst in den nächsten Kapiteln die eingebauten Serviceprogramme kennenlernen werden. Schalten Sie also den SMON ein und geben Sie das Programm 5 ein (ab $6000):

```
6000  PHA          mit diesen Befehlen retten wir Akku und Register auf
6001  TXA          den Stapel.
6002  PHA
6003  TYA
6004  PHA

6005  LDA #$7F     0111 1111 ist das in binär.
6007  STA $DD0D    Dadurch werden alle NMIs, die vom CIA 2 kommen
                   könnten, gesperrt. Erinnern Sie sich: Bit 7 ist Null
                   beim Schreiben, also Sperrfunktion.
600A  LDY $DD0D    Lesen des Registers 13 löscht dieses und zeigt uns,
                   ob die NMI-Anforderung von dort kam.
600D  BMI $601A    falls NMI-Anforderung vom CIA 2 kam, wird verzweigt
600F  LDA $D020    ansonsten kommt der NMI von der RESTORE-Taste,
                   und in den Akku wird die Rahmenfarbe eingeladen
6012  EOR #$0E     Ausgehend davon, daß als Rahmenfarbe 14 vorliegt,
                   wird diese exklusiv geORDERt zu Null. Ist die Rah-
                   menfarbe 0, dann wird sie wieder 14.
6014  STA $D020    Einschreiben des neuen Farbwertes
6017  JMP $FEBC    Sprung in den Rest der normalen NMI-Routine
601A  JMP $FE72    Sprung in die normale NMI-Routine für den Fall, daß
                   die Anforderung durch den NMI-CIA kam.
```

Programm 5. Ein kleines Testprogramm demonstriert die Wirkung einer Unterbrechung: Durch Drücken der RESTORE-Taste wird die Rahmenfarbe geändert.

Am besten speichern Sie nun das Programm ab und schalten dann mittels dem SMON-Kommando M 0318 die Anzeige der Bytes ab $0318 ein. Dort steht in den beiden ersten Speicherzellen 47 und FE. Mit dem Cursor fahren Sie in diese Zeile und ändern den Inhalt in 00 und 60, also unsere Programmstartadresse in der LSB/MSB-Form. Nach einem RETURN läuft nun jede NMI-Anforderung über unser Programm. Nun können Sie es ausprobieren, indem Sie mal die RESTORE-Taste drücken. Es genügt völlig, alleine diese Taste zu betätigen. Das wirkt – sichtbar durch die Änderung der Rahmenfarbe – in jedem Modus und jederzeit. Eine kleine Merkwürdigkeit ist, daß man manchmal etwas Geduld aufbringen muß, bis man die Wirkung sieht. Ich vermute, daß der NMI so schnell erledigt wird, daß sich mehrer NMIs pro Tastendruck ereignen. Man müßte sich noch eine kleine Routine überlegen, die die Wirkung etwas verzögert, denn 2 solche EOR-Kommandos nacheinander heben sich gegenseitig auf. Zum Schluß noch eine Aufstellung (Tabelle 20) mit den besprochenen Befehlen.

56. Der normale Verlauf eines IRQ

Neulich hatten wir bereits festgestellt, daß eine IRQ-Anforderung (nach dem Retten des Programmzählers und des Prozessorstatus-Registers, sowie dem Setzen der I-Flagge) den Inhalt des Vektors $FFFE/FFFF in den Programmzähler holt. Dort steht die Adresse $FF48(dez. 65352) und deshalb startet nun das dort im ROM verankerte Programm, welches wir uns nun im einzelnen ansehen werden (alle Adressen als Dezimalzahlen, in Bild 41 finden Sie das Flußdiagramm dazu)

Befehlswort	Adressierung	Bytezahl	Code Hex	Code Dez	Taktcyclen	Beeinflussung von Flaggen
CLI	implizit	1	58	88	2	I-Flagge
SEI	implizit	1	78	120	2	I-Flagge
RTI	implizit	1	40	64	6	alle Flaggen
BRK	implizit	1	00	0	7	B-Flagge vor dem Schieben auf den Stapel, I-Flagge danach

Tabelle 20. Die Daten zu den letzten Assembler-Befehlen

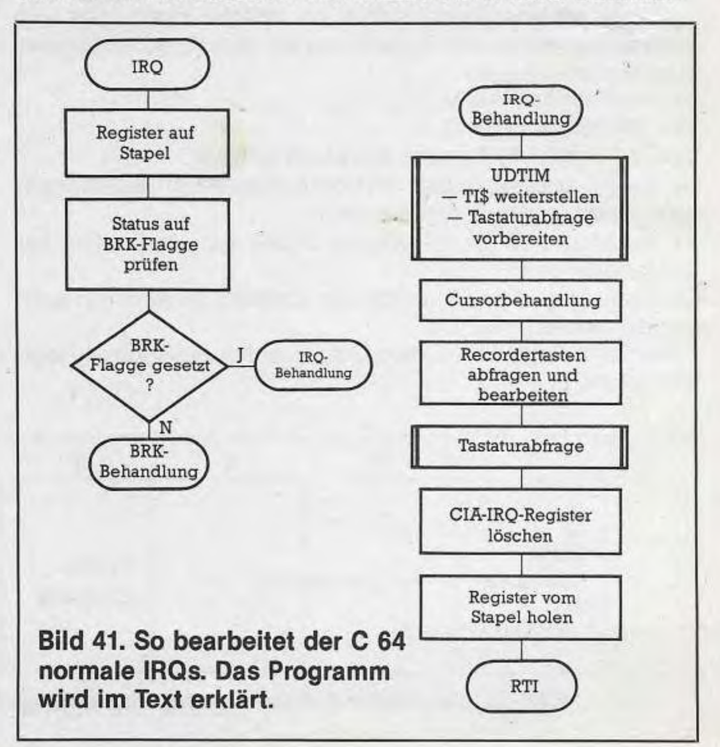


Bild 41. So bearbeitet der C 64 normale IRQs. Das Programm wird im Text erklärt.

65352	**PHA**	Zunächst werden der Akku und
	TXA	die Register X und Y auf den Sta-
	PHA	pel geschoben
	TYA	
	PHA	

Trickreich sind die beiden folgenden Befehle, mit denen das zu Beginn durch die CPU gerettete Statusregister gelesen wird:

	TSX	Stapelzeiger ins X- Register
	LDA 260,X	Einladen des Status-Registers

Nun wird geprüft, ob die BRK-Flagge gesetzt ist. Wenn das der Fall ist, dann ist der Auslöser ein BRK gewesen, ansonsten ein IRQ:

	AND #16	Isolieren der BRK-Flagge
	BEQ 65368	Wenn keine BRK-Flagge, dann überspringen des nächsten Befehls.
65365	**JMP (790)**	Falls BRK
65368	**JMP (788)**	Falls IRQ

Den vorletzten Sprungbefehl werden wir bei der BRK-Behandlung verfolgen. Interessant für uns ist jetzt der indirekte Sprung bei 65368. Der Vektor 788/789 ($314/315) liegt im RAM! Damit können wir ihn auf eigene Routinen verstellen. Genau hier ist der Ansatzpunkt für nahezu alle Eingriffe in die Unterbrechungsbehandlung. Der voreingestellte Wert in diesem Vektor ist die Adresse 59953 ($EA31). Das dort angesiedelte Programm wird im Normalfall 60 mal in der Sekunde ausgeführt:

59953	**JSR 65514**	Das ist ein Kernel-Sprungbefehl zur Routine UDTIM bei 63131.

In diesem Unterprogramm wird zuerst die Uhr TI$ weitergestellt und dann die Tastaturabfrage vorbereitet.

59956 bis 60000		In diesem Programmteil erfolgt die Cursorbehandlung.
60001 bis 60026		Anschließend wird abgefragt, ob eine Recordertaste gedrückt ist und entsprechende Flaggen bearbeitet.
60027	**JSR 60039**	Dieses Unterprogramm dient zur Tastaturabfrage.

Auch in dieser Routine tritt übrigens ein indirekter Sprung nach einem RAM-Vektor auf (655/656 = $28F/290), der normalerweise auf 60232 zeigt, aber auch auf eine eigene Routine verbogen werden könnte.

Enthalten in der Tastaturabfrage ist auch die Überprüfung der RUN/STOP-Taste, die aber nur zusammen mit den in dem UDTIM-Aufruf voreingestellten Flaggen funktioniert. Deshalb wird das Abschalten der RUN/STOP-Taste im allgemeinen dadurch durchgeführt, daß man den IRQ-Vektor auf 59956 stellt und damit den ersten JSR-Befehl überspringt. Allerdings wird auf diese Weise auch die TI$-Uhr nicht weitergestellt.

60030	**LDA 56333**	Das ist das Unterbrechungs-Kontrollregister des IRQ-CIA, das hier durch Auslesen gelöscht wird.

Den Abschluß der IRQ-Routine bildet nun noch das Zurückschreiben der Register:

60033	**PLA**	Zurückholen des
	TAY	Y- und
	PLA	
	TAX	des X-Registers
	PLA	sowie des Akku.
60038	**RTI**	Damit kehrt der Computer zu dem durch den IRQ unterbrochenen Programm zurück.

Somit hätten wir's. Nun können wir je nach Bedarf entscheiden, welche von diesen Servicetätigkeiten wir bei einem eigenen IRQ-Programm brauchen: Die Uhr TI$, die Cursorbehandlung, die Abfrage der Recordertasten und die Tastaturabfrage.

Sehen wir uns nun an, was geschieht, wenn ein BRK-Kommando der Auslöser war.

57. BRK-Unterbrechung

Wir hatten vorhin am Scheideweg zwischen IRQ und BRK den letzteren links liegen gelassen. Normalerweise verwendet man beim Programmieren in Assembler ja ein Software-Instrument wie zum Beispiel den SMON, der so gebaut ist, daß der BRK-Vektor, welchen wir vorhin kennengelernt haben ($316/317 = 790/791) auf die Registeranzeige weist. Was geschieht eigentlich, wenn der BRK-Vektor unverändert bleibt, so also, wie er im Einschaltzustand des Computers vorliegt?

Dann zeigt er auf die Adresse 65126 ($FE66), wo ein Teil der NMI-Routine zu finden ist (Siehe auch das Flußdiagramm in Bild 42):

65126	**JSR 64789**	Sprung ins Programm RESTOR, in dem alle Vektoren (788-819) gemäß einer ROM-Liste auf ihre Ausgangswerte gesetzt werden.
	JSR 64931	Sprung in das Programm I/O-RESET.

In diesem Programm werden die beiden CIAs auf die Anfangswerte gestellt.

	JSR 58648	Sprung in ein Programm, welches zuerst den VIC-II-Chip initialisiert, dann einen Bildschirmeditor-RESET durchführt.Nach Beenden dieser Routine ist der Bildschirm gelöscht.
	JMP (40962)	

Mit diesem indirekten Sprung ist die BRK-Unterbrechung beendet. Man sieht aber jetzt schon deutlich, daß es sich hier nicht um eine Unterbrechung im eigentlichen Sinn handelt,

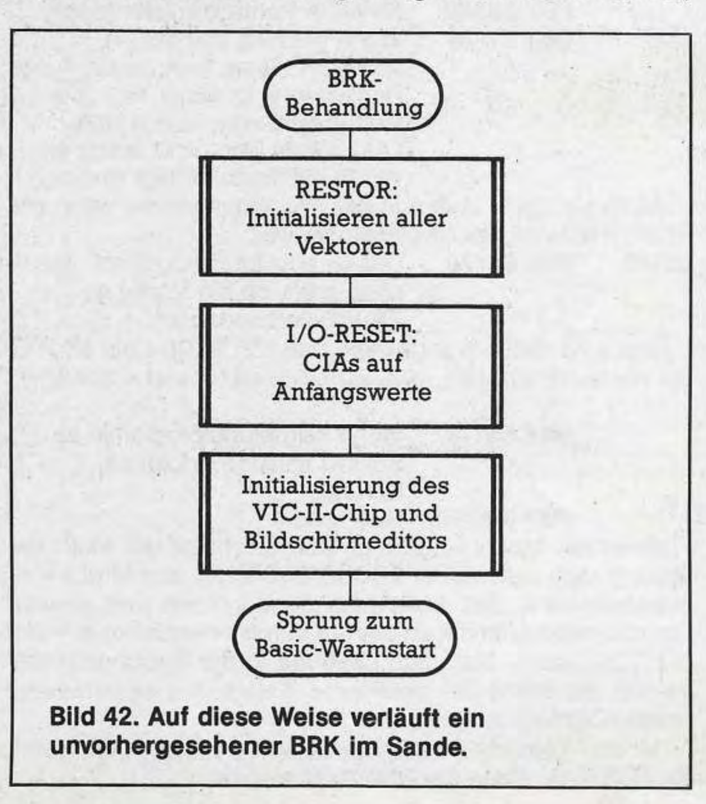


Bild 42. Auf diese Weise verläuft ein unvorhergesehener BRK im Sande.

vielmehr um einen Abbruch. In 40962/40963 steht die Adresse des Basic-Warmstarts (58235). Danach befindet sich der Computer im READY-Zustand in der Eingabe-Warteschleife.

Das Zurückholen der Register und ein RTI erübrigt sich hier, weil ohnehin viele Werte aus dem unterbrochenen Programm inzwischen weitgehend zerstört sind und alle Unterbrechungskontrollregister (CIAs und VIC-II-Chip) neu belegt wurden. Ein unkontrollierter BRK hat also recht fatale Folgen!

58. Was macht ein NMI?

Wenden wir uns nun der Firmware zu, die zur Bearbeitung eines NMI vorgesehen ist (Dazu sehen Sie sich bitte in Bild 43 das Flußdiagramm an).

In den letzten Kapiteln erfuhren wir, daß auch für diese Unterbrechung am Ende des Speichers ein Vektor vorhanden ist, nämlich $FFFA/FFFB (65530/65531). Dort steht die Adresse 65091 ($FE43), die nun in den Programmzähler gelangt und damit startet das folgende Programm:

65091	**SEI**	Unterbrechungen niedrigerer Priorität werden gesperrt.
	JMP (792)	

Das ist nun wieder ein für uns sehr interessanter Vektor 792/793 ($318/319), der — weil er im RAM-Bereich liegt — verstellbar ist. Genau das haben wir am Ende der letzten Folge getan mittels des M-Kommandos von SMON um den NMI zu testen, den wir mit der RESTORE-Taste ausgelöst haben. Der vorher eingestellte Wert in diesem Vektor ist die Adresse 65095 ($FE47), also direkt der nächste Befehl nach dem indirekten Sprungbefehl.

65095	**PHA**	Ebenso wie vorhin beim IRQ wer-
	TXA	den hier die Inhalte des Akku und
	PHA	der
	TYA	Register auf den Stapel
	PHA	geschoben.
	LDA #127	das ist binär 01111111.
	STA 56589	Sperrt alle weiteren NMI-Anforderungen
	LDY 56589	NMI-CIA Kontrollregister laden.
	BMI 65138	Wenn der NMI von der RESTORE-Taste kam, ist Bit 7 des Registers = 0, sonst = 1 (bei NMI-Anforderung durch NMI-CIA). Wenn also nicht durch die RESTORE-Taste, erfolgt Sprung.

An dieser Stelle läuft nun das Programm weiter, wenn die RESTORE-Taste der NMI-Auslöser war:

65110	**JSR 64770**	Das ist ein Unterprogramm, welches prüft, ob ein Modul ab $8000 vorhanden ist.

Dies wird dadurch angezeigt, daß von $8004 bis $8008 die Werte stehen: 195, 194, 205, 56, 48 (das ist »CBM80«).

	BNE 65118	Wenn kein Modulprogramm ab $8000 vorliegt, erfolgt ein Sprung.
	JMP (32770)	Falls Modul.

Wenn ein Modul angezeigt wurde, erfolgt der indirekte Sprung nach dem Vektor $8002/8003, der vom Modul vorgegeben wird. Das kann man auch nutzen, um eigene Maschinenprogramme zu starten durch einen Druck auf die RESTORE-Taste. Man muß dann nur in die Speicherstellen $8002 bis 8008 die geforderte Zieladresse beziehungsweise »CBM80« schreiben.

Der nun folgende Abschnitt wird nur angesprungen, wenn die RESTORE-Taste der NMI-Auslöser war:

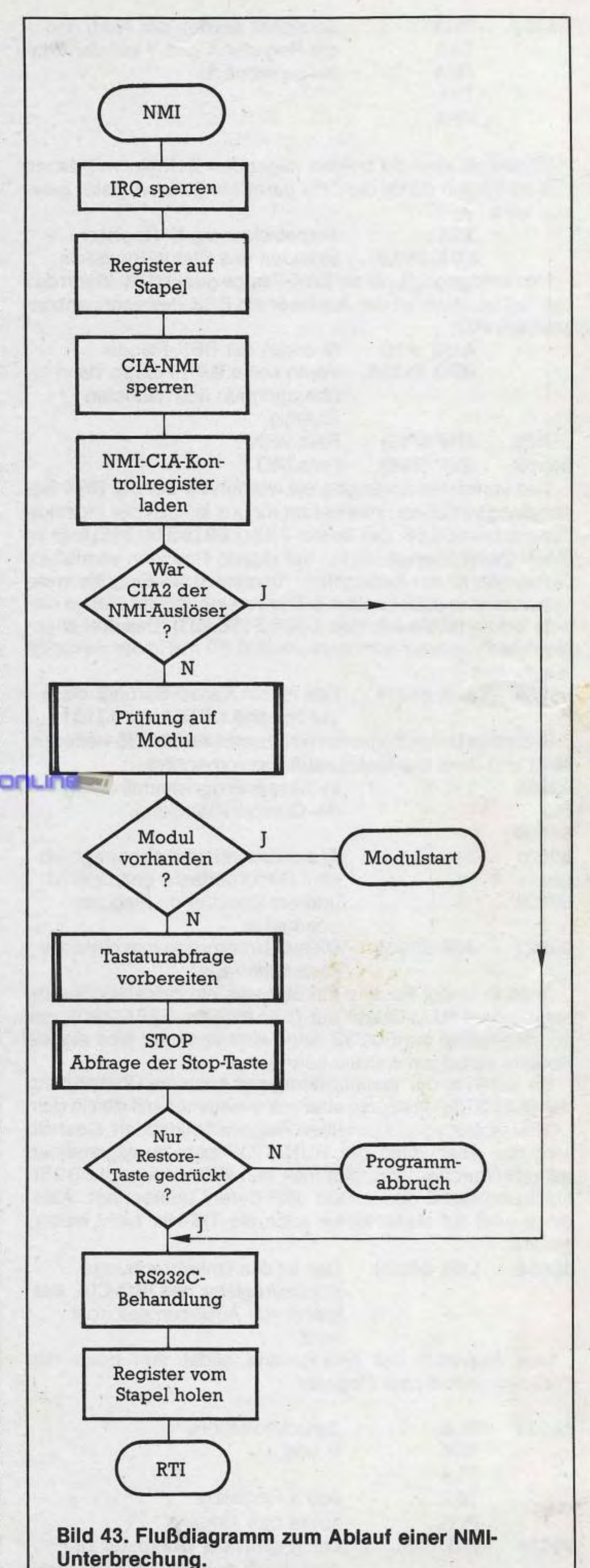


Bild 43. Flußdiagramm zum Ablauf einer NMI-Unterbrechung.

65118 **JSR 63164** Das ist ein Programmteil, der auch schon von der IRQ-Routine (nach dem Weiterstellen von TI$) durchlaufen wird. Hier werden einige Voreinstellungen für die Tastaturabfrage erledigt, die insbesondere die RUN/STOP-Taste betreffen.

JSR 65505 Kernelroutine STOP.

Dort befindet sich ein indirekter Sprung über den Vektor 808/809 ($328/329), also auch ein verstellbarer RAM-Vektor. Im Normalfall zeigt dieser Vektor auf 63213 ($F6ED). Dort wird geprüft, ob die RUN/STOP-Taste gedrückt ist. Eine andere Methode zum Ausschalten des RUN/STOP bietet sich hier an, die die Uhr TI$ ungeschoren läßt.

BNE 65138 Falls nur die RESTORE-Taste (also ohne RUN/STOP) gedrückt ist, erfolgt nun ein Sprung.

Waren aber sowohl die RUN/STOP- als auch die RESTORE-Taste gedrückt, dann folgt nun ein Programmabbruch, der uns schon von BRK her bekannt ist. Hier wie dort endet das Ganze dann mit dem Reset der I/O-Bausteine, des VIC-II-Chips, der Vektoren, des Bildschirmeditors und das Ergebnis ist ein Basic-Warmstart.

Ab 65138 befindet sich der Rest der NMI-Routine, auf die das Programm läuft, wenn

1) die NMI-Anforderung nicht von der RESTORE-Taste kommt oder

2) zwar von dieser Taste kommt, aber die RUN/STOP-Taste nicht gedrückt ist.

65138 bis 65211 Dieser ganze Abschnitt ist zur Behandlung der RS232C-Schnittstelle eingerichtet.

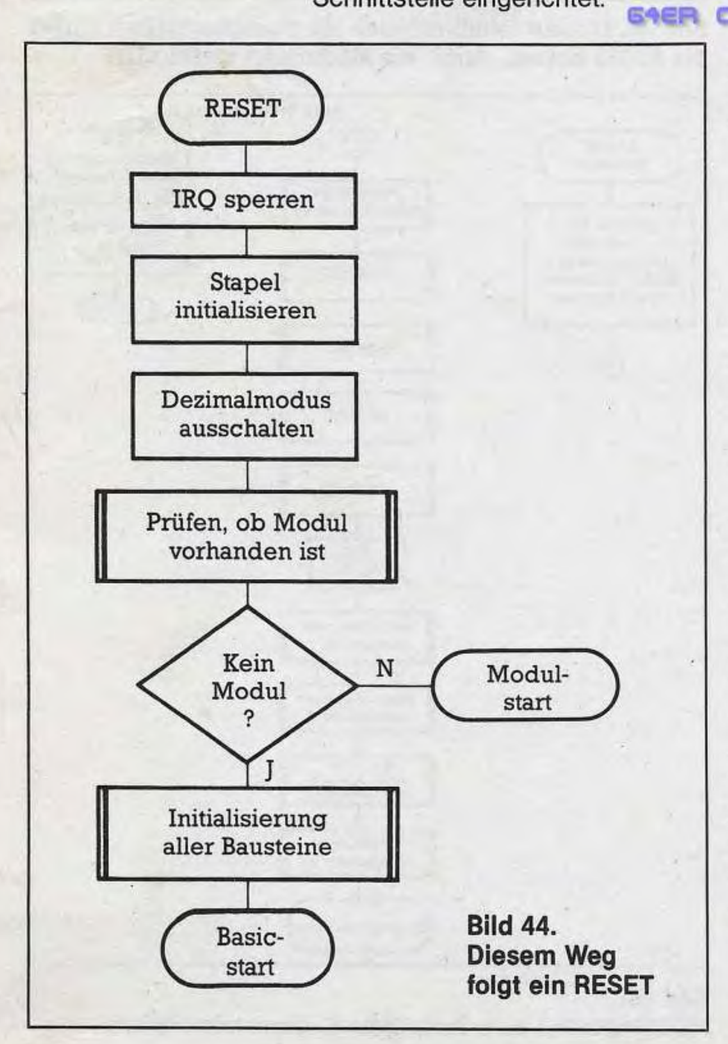


Bild 44. Diesem Weg folgt ein RESET

65212 **PLA** Abschluß des NMI durch Rückschreiben des Akku und der Register vom Stapel
TAY
PLA
TAX
PLA
65217 **RTI** Rückkehr zum unterbrochenen Programm.

Wenn Sie sich nun mal unser kleines Demo-Programm aus Kapitel 55 ansehen, dann werden Sie feststellen, daß der Programmteil bis $600E lediglich den ersten Teil der normalen NMI-Routine kopiert. Die Prüfung auf das Modul und die RUN/STOP-Taste werden übersprungen. Statt dessen erfolgt nach der Abarbeitung des für die RESTORE-Taste gebauten Programmes das Ende der NMI-Routine ($FEBC = 65212). Im anderen Fall, wenn also die RESTORE-Taste nicht der Auslöser des NMI war, wird in die normale Routine ab 65138 eingemündet.

59. Eigentlich keine Unterbrechung: RESET

Weil wir alle Unterbrechungen hier bearbeiten wollen, soll auch der RESET angesprochen werden. Es handelt sich dabei aber nicht um eine Unterbrechung im bisher definierten Sinn. Mir fällt allerdings kein Platz ein, wo der RESET besser hinpassen würde. Ähnlich wie bei NMI und IRQ wird auch hier ein Vektorinhalt in den Programmzähler geladen, der in den höchsten Speicheradressen zu finden ist (Auch hierzu wieder ein Flußdiagramm in Bild 44).

Dieser Vektor liegt in $FFFC/FFFD. Der Inhalt ist die Adresse 64738 ($FCE2) und genau dort geht das Programm dann weiter:

64738 **LDX #255** Im ersten Teil wird der Stapelspeicher initialisiert.
SEI Verhindern von IRQ
TXS Stapelzeiger auf $FF
CLD Dezimal-Modus ausschalten (falls er eingeschaltet war).
JSR 64770 Das ist wieder das Unterprogramm, das auf ein Modul prüft.

Hier ergibt sich die Möglichkeit, auch beim RESET einzugreifen, indem man die Kennung CBM80 an die abgefragten Orte packt.

BNE 64751 Falls kein Modul, erfolgt Sprung.
64748 **JMP(32768)**

Dieser indirekte Sprung erfolgt nach dem Vektorinhalt von $8000/8001 = 32768/32769. Das ist ein anderer Vektor als wir ihn vorhin beim NMI hatten (dort war es $8002/8003 = 32770/32771). So kann ein anderer Programmteil angesteuert werden als durch den NMI, was übrigens auch dringend erforderlich ist, weil der Stapelzeiger zerstört wurde.

64751 Hier läuft das Programm weiter, falls keine Modulkennung erkannt wurde.

Der ganze Rest dient dem Versetzen des Computers in den Einschaltzustand. Allerdings bin ich davon überzeugt, daß noch irgendein Unterschied bestehen muß zwischen dem einfachen Aus- und wieder Anschalten des Computers und einem RESET. Es hat sich nämlich bei einigen Programmen gezeigt, daß sie nach einem RESET fehlerhafte Verläufe nehmen können, was nach einem totalen Aus- und wieder Anschalten nicht zu beobachten war. Der Grund für diesen Unterschied liegt (für mich) noch im Dunkeln. Vielleicht weiß das ja jemand von Ihnen. Dann schreiben Sie doch mal!

60. Die Sache mit dem Modulstart

Sowohl beim RESET als auch beim NMI haben wir festgestellt, daß der Modulstart-Bereich ab $8000 eine besondere Rolle spielt. In Bild 45 finden Sie nochmal zusammengefaßt, was sich dort findet wenn ein Modul vorhanden ist.

Wir wollen im folgenden Beispielprogramm (Programm 6) ein Modul simulieren, indem wir den SMON mittels des RESET anspringen. Der NMI — also die RUN/STOP-RESTORE-Tastenkombination — soll dabei wirkungslos gemacht werden.

Bild 46 zeigt ein Flußdiagramm dieses Beispielprogrammes:

Achten Sie bitte darauf, daß Sie nach dem Eintippen des Programmes abspeichern und — natürlich — daß die SMON-Version ab $C000 im Speicher vorliegt. Wenn Sie nun mal die RESTORE-Taste — oder RUN/STOP und RESTORE — drücken, passiert offensichtlich nichts. Das liegt daran, daß unser Programm lediglich die auf den Stapel gelegten Register wieder zurückholt und aus der Unterbrechung mit RTI ins normale Geschehen zurückkehrt.

Haben Sie einen RESET-Taster eingebaut? Dann drücken Sie doch mal drauf. Zunächst erkennen Sie den normalen RESET-Verlauf. Dann meldet sich aber nicht wie gewohnt die Nachricht CBM-Basic..., sondern der SMON mit einer Registeranzeige. Das RESET-Programm ab $602E folgt dem Firmware-Programm. Auf diese Weise (und mittels eines AUTOSTART) sichern sich Softwarehäuser manchmal gegen unbefugtes Kopieren ihrer Programme.

```
100  -            .LI 1,3
110  -            .BA $8000
112  -;
114  -;********************************
116  -;  MODULSIMULATION
117  -;********************************
118  -;
120  -            .EQ INITCZ=$E3BF
130  -            .EQ INITMS=$E422
140  -            .EQ INITV=$E453
150  -            .EQ SCREENCLR=$E544
160  -            .EQ RAMTEST=$FD50
170  -            .EQ IORESET=$FDA3
180  -            .EQ TVTAKT=$FF5B
190  -            .EQ RESTOR=$FF8A
192  -;
194  -;  ****** MODULKENNUNG UND -VEKTOREN ****
196  -;
200  -            .WO RESET,NMI
210  -            .BY $C3,$C2,$CD,$38,$30
212  -;
214  -;******   RESET-PROGRAMM ********
216  -;
220  -RESET       STX $D016        ;RESET-BIT
230  -            JSR IORESET
240  -            JSR RAMTEST
250  -            JSR RESTOR
260  -            JSR TVTAKT
270  -            CLI
280  -            JSR INITV
290  -            JSR INITCZ
300  -            JSR INITMS
310  -            JSR SCREENCLR
320  -            JMP $C000        ;SPRUNG IN SMON
322  -;
324  -; ** NMI-PROGRAMM (RESTORE-TASTE) **
330  -NMI         PLA
340  -            TAY
350  -            PLA
360  -            TAX
370  -            PLA
380  -            RTI
390  -            .SY 1,4
```

Programm 6. Simulation eines Moduls

61. Nutzung der Unterbrechungen

Sowohl was die Hardware als auch die Firmware für die Unterbrechungsbehandlung angeht, haben wir nun einen guten Überblick gewonnen. Es ist jetzt an der Zeit, daß wir uns ansehen, auf welche Weise man dieses Reservoir an vielfältigen Möglichkeiten für sich nutzen kann. Dazu soll uns ein Überblick dienen:

I) Auslösung der Unterbrechung durch Hardware-Einwirkungen.

Da hätten wir beispielsweise den Userport oder den Expansion-Port, über die wir per CIAs Unterbrechungen anfordern können. Um es gleich zu sagen: Damit werden wir uns nicht auseinandersetzen. Meine Kenntnisse auf diesem Gebiet sind zu dünn. Aber vielleicht verstehen Sie das auch mal als Aufforderung, Ihre Versuche dazu anderen zu offenbaren? Also: Schreiben Sie doch mal!

II) Unterbrechungsauslösung per Software:

Damit haben wir immer noch ein weites Feld von Möglichkeiten vor uns:

IIa) Vorgesehene Nutzungen des IRQ
— mittels des VIC-II-Chips.

Speicherplatz ($)	8000	8001	8002	8003	8004	8005	8006	8007	8008
Inhalt	LSB	MSB	LSB	MSB	C	B	M	8	0
	RESET-Vektor		NMI-Vektor						

Bild 45. Diesen Inhalt müssen die Speicherstellen $8000 bis $8008 haben, damit ein Modulstart stattfindet.

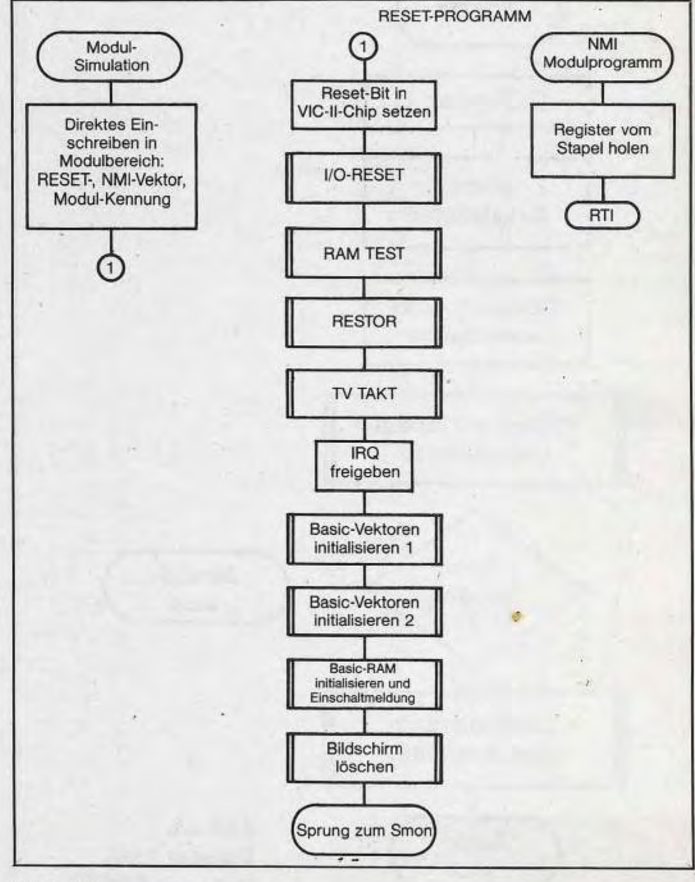


Bild 46.
Flußdiagramm zum Programm 6, Modulsimulation.

Da können wir uns auf den Rasterzeileninterrupt, die Sprite/Hintergrund- oder die Sprite/Sprite-Kollision stützen.
— oder mit Hilfe des CIA1

Da ist es vor allem der 60mal pro Sekunde auftretende Timer A-Unterlauf, der uns interessieren soll.

IIb) Vorgesehene Nutzungen des NMI
— CIA2: Läßt man die RS232C-Schnittstellenbehandlung außer acht, dann gibt es keine vorgesehene Nutzung.
— RESTORE: Zusammen mit der RUN/STOP-Taste kann man die vorgegebene Routine verändern, wie wir es schon in einigen Beispielen gezeigt haben.

Wir können außerdem noch unterscheiden zwischen Nutzungen, die periodisch stattfinden sollen (zum Beispiel eine spezielle Tastaturabfrage) und solchen, die stochastisch (= zufallsabhängig) oder willkürlich erfolgen (zum Beispiel Drücken der RESTORE-Taste). Beides ist auch durchführbar bei:

IIc) Nicht vorgesehene Nutzung der Unterbrechungen.

Da bietet sich vor allem der meistens völlig brach liegende CIA2 an mit seinen beiden Timern und der Alarmfunktion.

Wenn Sie aber erst einmal vertraut sind mit der Unterbrechungs-Programmierung und auch etwas Zeit zum Tüfteln investieren, finden Sie bestimmt noch eine ganze Menge weiterer Möglichkeiten.

Bei mehreren gleichartigen Unterbrechungsanforderungen (zum Beispiel IRQs) muß noch ein Weg gefunden werden, wie zwischen den dann vielleicht anfallenden unterschiedlichen Service-Routinen differenziert werden kann. Denkbar wären beispielsweise Aufgabenstellungen wie:

Jeder 3. Timer-IRQ soll den Joystick abfragen, oder RESTORE+h soll den Hilfsbildschirm zeigen, RESTORE+z soll den aktuellen Bildschirm wieder restaurieren, etc.


```
PROGRAMM 7

,6000  78        SEI
,6001  A9 28     LDA #28
,6003  8D 14 03  STA 0314
,6006  A9 60     LDA #60
,6008  8D 15 03  STA 0315
,600B  A9 F8     LDA #F8
,600D  8D 12 D0  STA D012
,6010  AD 11 D0  LDA D011
,6013  29 7F     AND #7F
,6015  8D 11 D0  STA D011
,6018  A9 81     LDA #81
,601A  8D 1A D0  STA D01A
,601D  A9 00     LDA #00
,601F  8D 20 D0  STA D020
,6022  A9 04     LDA #04
,6024  85 02     STA   02
,6026  58        CLI
,6027  60        RTS
--------------------------
,6028  AD 19 D0  LDA D019
,602B  8D 19 D0  STA D019
,602E  30 07     BMI 6037
,6030  AD 0D DC  LDA DC0D
,6033  58        CLI
,6034  4C 31 EA  JMP EA31
--------------------------
,6037  AD 12 D0  LDA D012
,603A  C9 F8     CMP #F8
,603C  B0 11     BCS 604F
,603E  18        CLC
,603F  65 02     ADC   02
,6041  8D 12 D0  STA D012
,6044  A0 03     LDY #03
,6046  88        DEY
,6047  D0 FD     BNE 6046
,6049  EE 20 D0  INC D020
,604C  4C 81 EA  JMP EA81
--------------------------
,604F  A9 00     LDA #00
,6051  8D 20 D0  STA D020
,6054  A9 32     LDA #32
,6056  8D 12 D0  STA D012
,6059  4C 81 EA  JMP EA81
--------------------------
,605C  78        SEI
,605D  A9 00     LDA #00
,605F  8D 1A D0  STA D01A
,6062  A9 31     LDA #31
,6064  8D 14 03  STA 0314
,6067  A9 EA     LDA #EA
,6069  8D 15 03  STA 0315
,606C  A9 0E     LDA #0E
,606E  8D 20 D0  STA D020
,6071  58        CLI
,6072  60        RTS
--------------------------
```

Programm 7. Das im Artikel entwickelte Programm auf einen Blick

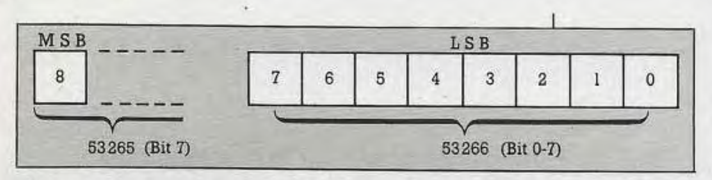


Bild 47. So sieht das 9-Bit-Register im VIC-II-Chip aus, welches die Rasterzeilen mitzählt.

Sie sehen, eine große Menge Arbeit wartet auf uns. Nicht zu allen Möglichkeiten werde ich hier Beispielprogramme zeigen. Außerdem dürfen die dann auch nicht zu undurchsichtig sein und man sollte möglichst den Erfolg eines solchen Demo-Programmes auf dem Bildschirm erkennen können. Trotzdem hoffe ich, daß die nachfolgend und in den nächsten Kapiteln gezeigten Programmlösungen ausreichen, Ihnen die Unterbrechungs-Behandlung mit eigenen Routinen durchschaubar zu machen. Ich will Ihnen aber nicht verschweigen, daß auch mir noch längst nicht alle Geheimnisse der Unterbrechungsprogrammierung offenbar geworden sind. Oft finde ich mich unversehens in Programm-Sackgassen wieder. Das soll Ihnen als kleiner Trost dienen, wenn Sie mal nach dem 1001. Absturz müde und mit rauchendem Kopf vor Ihrem Commodore-Ungeheuer sitzen.

62. Ein Programm zum VIC-II-IRQ

Sehr schöne Effekte lassen sich durch eine periodische IRQ-Anforderung per Rasterzeileninterrupt mittels des VIC-II-Chip erzielen. Deshalb ist sowas auch ein beliebtes Objekt für Demos von Unterbrechungsprogrammen. Als Ziel setzen wir uns, einen Bildschirm zu konstruieren, dessen Rahmen in allen Farben schillert.

Leser der Grafikserie werden diese Möglichkeit des VIC-II-Chip schon kennen: Man kann dem Kathodenstrahl, der über den Monitor huscht, um das Bild zu erzeugen, über zwei Register folgen, die Rasterregister, wo jede Rasterzeile mitgezählt wird. Ohne an dieser Stelle allzusehr auf die Einzelheiten einzugehen, soll hier nur bemerkt werden, daß die Numerierung dabei etwa von 0 bis 280 geht, weil auch der Rahmen und nicht sichtbare Teile des Bildschirmes vom Strahl überstrichen werden. Wo das Textfeld anfängt, ist von Monitor zu Monitor (oder Fernseher) etwas unterschiedlich. Bei mir beginnt es oben in Rasterzeile 50 und endet unten bei Zeile 248. Sollten die im Beispielprogramm 7 (Programm 7) nachher voreingestellten Randwerte bei Ihnen also anders sein, können Sie sie durch einige später noch angegebenen POKEs ändern. Die beiden Rasterzeilenregister sind:

$D012 (53266)
$D011 (53265)

Von $D011 allerdings ist nur das Bit 7 als msb der Rasterzeilenzahl für uns von Bedeutung. Bild 47 soll diese Belegung deutlich machen:

Das Interessante an diesen Registern ist nun, daß man auch in sie schreiben kann. Die auf diese Weise festgelegte Rasterzeile ist dann der Auslöser des IRQ, falls dieser im Interrupt-enable-Register $D01A freigegeben wurde (das kennen wir noch aus Kapitel 51).

Damit kann also unsere primäre Unterbrechungsquelle (der VIC-II-Chip) programmiert werden. Halten wir die zwei Schritte dazu nochmal fest:

1) Rasterzeile festlegen, bei der ein IRQ ausgelöst werden soll, durch Einschreiben in die Register $D012 und Bit 7 von $D011.

2) Freigeben des Rasterzeileninterrupts durch Einschreiben von 1000 0001 in das Interrupt-enable-Register $D01A.

Der nächste Schritt betrifft die Bearbeitung des IRQ durch die CPU. Wie wir vorhin sahen, springt das Programm beim IRQ mittels eines indirekten Sprunges, der auf den Vektor 788/9 ($314/5) zugreift. Dieser Vektor muß nun auf die eigene Routine verbogen werden, also:

3) Vektor $314/5 auf die IRQ-Service-Routine richten.

Damit wären alle Vorbereitungen getroffen. Der Rest liegt nun ganz bei uns — beziehungsweise bei dem von uns zu schreibenden Service-Programm. Als Bild 48 finden Sie ein Flußdiagramm unseres Beispielprogrammes 7.

Gehen wir nun an die Realisierung. Zunächst also die Initialisierung, die wir bei $6000 (also durch SYS 24576 zu starten) beginnen lassen:

```
6000  SEI          Sperren von IRQs
Schritt 3:
6001  LDA #$28     LSB der IRQ-Routine
6003  STA 0314     in IRQ-Vektor-LSB
6006  LDA #$60     MSB der IRQ-Routine
6008  STA 0315     in IRQ-Vektor-MSB
Schritt 1:
600B  LDA #$F8     Rasterzeile, bei der das Textfen-
                   ster endet. Von da an soll der
                   Rahmen schwarz sein.
600D  STA D012     in Rasterzeilen-Register (LSB)
                   schreiben.
6010  LDA D011     Register mit dem msb des
                   Rasterzeilenzählers
6013  AND #$7F     0111 1111 löscht das Bit7
6015  STA D011     Zurückschreiben. Damit ist die
                   Rasterzeile, die den IRQ auslösen
                   soll, festgelegt.
Schritt 2:
6018  LDA #$81     1000 0001 wird nun
601A  STA D01A     ins IRQ-enable-Register
                   geschrieben, um den
                   Rasterzeilen-IRQ zuzulassen.
```

Festlegen einiger Startwerte:

```
601D  LDA #$00     Farbe schwarz
601F  STA D020     in Rahmen schreiben
6022  LDA #$04     Streifenbreite in
6024  STA 02       Merkregister schreiben.
6026  CLI          IRQ freigeben
6027  RTS          Ende der Initialisierung.
```

Von nun an laufen alle IRQs über unsere eigene Routine, die bei $6028 beginnt.

Zunächst müssen wir prüfen, ob die Unterbrechung vom VIC-II-Chip kommt oder vom CIA1:

```
6028  LDA D019     IRQ-Request-Register des VIC-II-
                   Chip (siehe Kapitel 51). Dort ist
                   Bit 7 gesetzt, wenn die Anforde-
                   rung vom VIC-II-Chip kam.
602B  STA D019     Zurückschreiben
602E  BMI 6037     Sprung, falls VIC-IRQ, sonst
                   CIA-IRQ.
```

Bearbeiten eines CIA-IRQ:

```
6030  LDA DC0D     Löschen des CIA1
                   Unterbrechungs-Kontrollregisters.
```

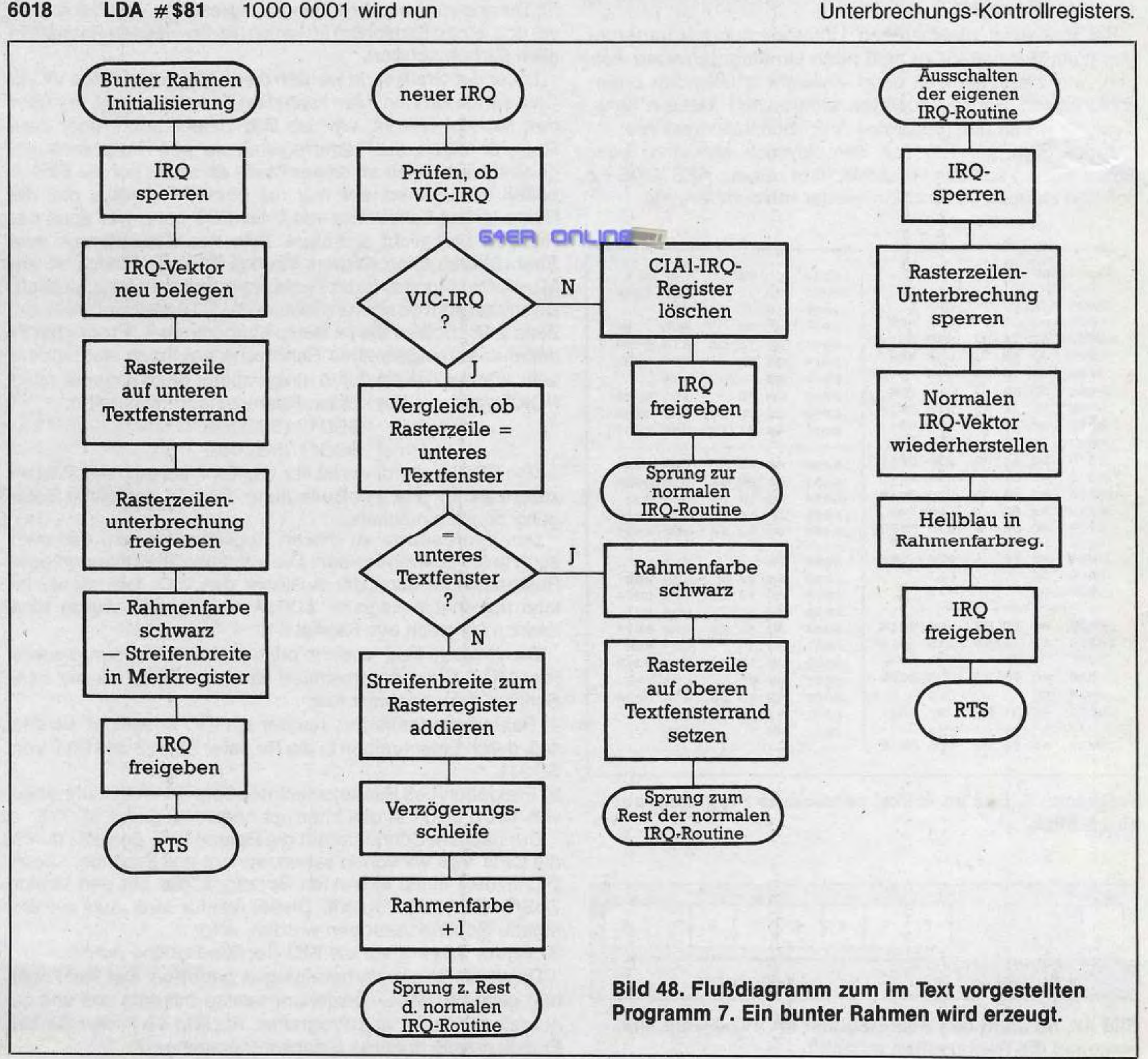


Bild 48. Flußdiagramm zum im Text vorgestellten Programm 7. Ein bunter Rahmen wird erzeugt.

```
6033   CLI         IRQ zulassen. Damit können
                   innerhalb eines CIA- IRQ auch
                   unsere VIC-IRQs geschehen.
6034   JMP EA31    Bearbeitung des CIA-IRQ durch
                   die normale Routine.
```

Unser Programm für VIC-II-IRQs:

```
6037   LDA D012    Rasterzeilen-Register laden um
                   festzustellen, welche Zeile den
                   IRQ auslöste.
603A   CMP #$F8    Vergleich mit Ende des Text-
                   fensters.
603C   BCS 604F    Wenn unterhalb des Textfensters,
                   Sprung.
```

Der folgende Programmteil ist wirksam, wenn der IRQ-Auslöser eine Zeile in Höhe des Textfensters war:

```
603E   CLC         Addition vorbereiten.
603F   ADC 02      Streifenbreite aus dem Merkregi-
                   ster addieren.
6041   STA D012    Neuen Wert in Rasterzeilen-
                   Register schreiben.
```

Damit wird eine neue Rasterzeile als IRQ-Auslöser festgelegt, die um die Streifenbreite tiefer liegt als die vorhergegangene.

Es folgt eine kleine Verzögerungsschleife, die aber nur zum Experimentieren eingebaut wurde:

```
6044   LDY #$03    Schleifen-Startwert
6046   DEY         Herunterzählen
6047   BNE 6046    NEXT Y, bis Y=0.
```

Ändern der Rahmenfarbe bis zum nächsten Raster-IRQ:

```
6049   INC D020    Farbcode+1. Wenn Code im
                   Rahmenfarbregister größer als 15
                   wird, fängt wieder Farbcode 0 an,
                   weil die Bits 5-7 keine Funktion
                   haben.
```


Abschließend erfolgt der Rücksprung in den Rest der normalen IRQ-Routine:

```
604C   JMP EA81    Siehe unsere Untersuchung der
                   IRQ-Firmware.
```

Damit ist der Rahmen in Höhe des Textfensters behandelt. Es schließt sich nun der Teil an, der die Rahmenbereiche unter- und oberhalb bearbeitet:

```
604F   LDA #$00    Farbcode schwarz
6051   STA D020    in Rahmenfarb-Register.
6054   LDA #$32    Rasterzeile, bei der oben das
                   Textfenster beginnt.
6056   STA D012    In Rasterzeilen-Register
                   schreiben
6059   JMP EA81    Abschluß durch Sprung zum
                   Ende der normalen IRQ- Routine.
```

Damit ist festgelegt, daß ober- und unterhalb des Textfensters die Rahmenfarbe schwarz wird.

Unsere eigene Routine ist jetzt abgeschlossen. Zum guten Ton gehört es, dem Benutzer auch die Möglichkeit zu öffnen, diese Routine wieder abzuschalten. Das erfolgt im letzten Programmteil, der mittels SYS24688 aktiviert werden kann:

```
605C   SEI         IRQ sperren
605D   LDA #$00    Raster-IRQ
605F   STA D01A    abschalten
6062   LDA #$31    IRQ-Vektor
6064   STA 0314    restaurieren
6067   LDA #$EA    auf den
6069   STA 0315    Normalwert.
606C   LDA #$0E    Farbcode hellblau
606E   STA D020    in Rahmenfarb-Register
                   schreiben
6071   CLI         IRQ zulassen
6072   RTS
```

Unser Programm ist komplett. Speichern Sie es bitte vor dem Starten ab. Nach dem SYS 24576 finden Sie einen hübschen bunten Rahmen vor, oberhalb und unterhalb des Textfensters ist er schwarz. Besonders gut — finde ich — sieht das Ganze aus, wenn man die Hintergrundfarbe des Textfensters auch auf Schwarz setzt. Das Programm erlaubt noch einige Experimente:

Durch POKE-Kommandos in die Speicherstelle 2 kann die aktuelle Streifenbreite variiert werden, durch POKEs in die Zelle 24645 der Startwert der Verzögerungsschleife. Probieren Sie's doch mal aus. Eine Erkenntnis werden Sie gewinnen: In der Unterbrechungs-Programmierung spielt die Zeit eine wichtige Rolle. Das zeigt sich auch, wenn man zum Beispiel Cursorbewegungen durchführt: Die Streifen fangen an zu wandern.

Weitere Möglichkeiten zum Experimentieren sind gegeben, wenn Sie die Rasterzeilen verändern, die den oberen und unteren Rand des Textfensters markieren:

Durch POKE 24661,Zahl verschieben Sie die obere, durch POKE 24635,X:POKE 24588,X die untere Rasterzeile, von der an alles schwarz ist. Wie schon vorhin erwähnt, habe ich im Programm diese Werte auf 50 beziehungsweise 248 fixiert, weil genau dort auf meinem Monitor das Textfenster liegt.

Mit diesem Beispiel und dem aus der Grafikserie sollte es Ihnen nun möglich sein, auch andere Unterbrechungsprogramme zu schreiben, die sich der Rasterzeilen-Unterbrechung per VIC-II-Chip bedienen. Eine Bemerkung sollte ich Ihnen noch auf den Weg Ihrer eigenen Versuche mitgeben: Der Elektronenstrahl, der über den Bildschirm saust und beim Erreichen des von uns bestimmten Rasterzeilenwertes zum Auslösen des IRQ führt, ist enorm schnell. Die Serviceprogramme dürfen deshalb nicht zu lang sein, sonst steht der nächste IRQ schon wieder an, bevor der vorangegangene bearbeitet ist.

63. Unterbrechungen mit den CIAs

Lassen Sie uns kurz rekapitulieren: Als primäre Unterbrechungsanforderer hatten wir drei Bausteine unseres Computers benannt, nämlich den VIC-II-Chip und die beiden CIA-Bausteine. CIA kommt von »Complex Interface Adapter« und ist die Bezeichnung für die beiden Ein- und Ausgabe-Bausteine, die den gesamten Verkehr zwischen dem zentralen Gehirn unseres C 64 und der Peripherie managen. Wir hatten bemerkt, daß ein CIA, der IRQ-CIA (Adressen von 56320 bis 56575), ausschließlich für die maskierbaren Unterbrechungen zuständig ist. Dazu gehören die 60mal pro Sekunde stattfindenden »Timer-Interrupts«, die die Cursorbehandlung, die TI$-Uhr, die Tastaturabfrage etc. bearbeiten. Der andere CIA, genannt NMI-CIA, (Adressenraum 56576-56831) ist nur für die nicht maskierbaren Unterbrechungen verantwortlich und wird bei normaler Nutzung des C 64 so gut wie nie eingesetzt. Ich gehe im folgenden davon aus, daß Sie keine RS232C-Schnittstelle in Ihren Computer eingesetzt haben. Sollte das aber der Fall sein, dann müßten Sie darauf achten, die folgenden Beispiele - die den NMI-CIA betreffen - ohne gleichzeitigen Betrieb dieser Schnittstelle anzuwenden, weil sich sonst Störungen ergeben könnten.

In Kapitel 52 haben wir uns ein Register (das Register 13, Interrupt-Kontrollregister) der CIAs schon genauer angesehen und auch die Unterschiede beider Bausteine festgestellt. Dort war dann die Rede von Timern, Echtzeituhren, Alarm-Funktionen etc. Was es damit auf sich hat und wie man diese Möglichkeiten nutzen kann, das soll nun unser Thema sein. Wir werden uns dazu alle Register der CIAs genauer ansehen, die für die von uns ausgewählten Unterbrechungsoptionen eine Rolle spielen. Dabei fallen einige unter den Tisch - das habe ich aber schon in Kapitel 52 angekündigt -, nämlich diejenigen, die mit dem Verkehr über den seriellen

Port, beziehungsweise über die RS232C-Schnittstelle, zu tun haben. Es bleibt dann anderen – kompetenteren – überlassen, darüber zu schreiben. Wie wäre es zum Beispiel mit Ihnen?

Auch so bleibt uns genug zu tun. In Tabelle 21 finden Sie zunächst eine Übersicht der von uns behandelten Register.

Sie sehen darin, daß jeder CIA über zwei sogenannte Timer (A und B) verfügt, sodann über die »Time of Day« (zu deutsch etwa »Tageszeit«) genannte Echtzeituhr mit vier Registern und schließlich noch über drei Kontrollregister, zu denen auch das schon erwähnte Register 13 gehört. Sehen wir uns zunächst die Timer an.

64. Die Timer der CIAs

Insgesamt verfügen wir über vier dieser Timer: Timer A und B im CIA1 und dasselbe nochmal im CIA2. Es handelt sich dabei um 16-Bit-Register, in die ein Startwert geschrieben werden kann, von dem an dann heruntergezählt wird. Jedesmal, wenn dann der Wert 0 unterschritten ist, gibt es für uns die Möglichkeit, bestimmte Ereignisse stattfinden zu lassen. Man kann diese Register unabhängig voneinander, aber auch kombiniert, benutzen. Ein Lesen des Registers liefert immer den momentan gerade aktuellen Wert. Ein Schreiben in das Register führt automatisch zum Festlegen eines Startwertes. Was an Optionen mit diesen Timern möglich ist, wird über Kontrollregister gesteuert. Das CRA (Register $ 0E) bezieht sich vor allem auf den Timer A, das CRB (Register $ 0F) auf Timer B. Die 16-Bit-Register werden – wie gewohnt – in der Form LSB/MSB betrieben. In den Timer A des CIA1 wird bei jedem I/O-Reset folgendes Wertepaar eingetragen:

56324	dezimal 37	LSB
56325	dezimal 64	MSB

Das entspricht einem Startwert von 16421. Im PAL-System hat der Quarz, der die Taktfrequenz bestimmt, eine Frequenz von 17.734472 MHz. Die Prozessorfrequenz errechnet sich daraus mittels Division durch 18 zu 985248.4 Hz (also etwas weniger als 1 MHz, was den europäischen C 64 langsamer macht als den amerikanischen, der etwas mehr als 1 MHz verwendet). Wenn mit dieser Geschwindigkeit der Timer heruntergezählt wird, erhält man alle $\frac{1}{60}$ Sekunden genau einen Unterlauf. Das ist der Weg, eine kontrollierte Zeitspanne durch den Timer zählen zu lassen. Sei X der gesuchte Startwert, der zu einer Spanne von T Sekunden führt, dann kann man X berechnen mittels:

X = 985248.4 * T

Der Integerwert von X ist dann in ein LSB und ein MSB zu teilen und in die Timer-Register einzutragen. Allerdings ergibt sich so eine natürliche Grenze. Die höchste durch 2 Byte darstellbare Zahl ist ja 65535. Wenn wir diesen Wert in den Timer schreiben, dann ist er alle 1/15 Sekunden auf 0 heruntergezählt. Für längere Zeiten ist aber vorgesorgt. Die beiden Timer A und B sind kombinierbar (wie, dazu kommen wir gleich noch) zu einem 32-Bit-Register. Die höchste Zahl X ist dann:

$$4\,294\,967\,296 = 2^{32}$$

Damit kann im Extremfall eine Herabzählzeit von 1 Stunde, 12 Minuten und zirka 40 Sekunden eingeplant werden, was für die meisten Zwecke ausreichen dürfte.

Möchten Sie also genau eine Sekunde Spielraum haben beim Herunterzählen, dann muß die Zahl 985248 als 4-Byte-Integer-Wert in die Speicher von Timer A und Timer B gebracht werden. Das führt dann zu den Werten 0, 15, 8, 160 (weil 985248 = 0*16777216 + 15*65536 + 8*256 + 160). 0 und 15 gelangen als MSB beziehungsweise LSB in Timer B (also Register 07 und 06), 8 und 160 sind MSB und LSB für den Timer A (Register 05 und 04). Sehen wir uns nun an, wie wir dem Computer sagen, was mit diesen Startwerten in den Timer-Registern geschehen soll. Die beiden Kontrollregister CRA und CRB beziehen sich weitgehend auf die gleichnamigen Timer. Im Bild 49 finden Sie das Register $0E, also CRA und in Bild 50 das andere Kontrollregister CRB ($0F):

Die Bedeutung der Bits 0 bis 4 ist – jeweils für den dazugehörigen Timer – identisch:

Bit 0 — an dieser Stelle führt zum sofortigen Anhalten des Timers. 1 in diesem Bit startet das Herunterzählen.

Bits 1 und 2 — Diese beiden Bits hängen mit dem externen Signalverkehr zusammen und werden von uns außer acht gelassen.

Bit 3 — Ist dieses Bit = 1, dann ist der sogenannte »One Shot«-Betrieb des Timers aktiv. Das bedeutet, daß vom Startwert an heruntergezählt wird bis auf Null. Es findet nun das programmierte Ereignis statt (zum Beispiel ein IRQ). Anschließend wird der Startwert wieder eingeladen und der Timer gestoppt.

Im Gegensatz dazu läuft der »Continuous«-Betrieb, wenn das Bit den Wert 0 enthält. Dabei geschieht zunächst dasselbe wie beim One Shot Modus, der Timer wird aber nicht angehalten, sondern der ganze Vorgang wiederholt sich in einer Endlosschleife.

Bit 4 — Ein Hineinschreiben einer 1 in dieses Bit erzeugt ein sofortiges Neuladen der Timer-Register mit dem Startwert. Dabei ist es gleich-

Register Nr. ($)	Adresse (dez.) CIA-1	CIA-2	Name	Funktion	
04	56324	56580	TALO	TIMER A	LSB
05	56325	56581	TAHI	TIMER A	MSB
06	56326	56582	TBLO	TIMER B	LSB
07	56327	56583	TBHI	TIMER B	MSB
08	56328	56584	TOD10THS	$\frac{1}{10}$-Sekunden-Register	
09	56329	56585	TODSEC	Sekunden-Register	
0A	56330	56586	TODMIN	Minuten-Register	
0B	56331	56587	TODHR	Stunden-Register, AM/PM-Flagge	
0D	56333	56589	JCR	Unterbrechungs-Kontrollregister	
0E	56334	56590	CRA	Kontrollregister A	
0F	56335	56591	CRB	Kontrollregister B	

Tabelle 21. Die wichtigen Register der beiden CIAs.

7	6	5	4	3	2	1	0
TODIN 50Hz 60 Hz	externer Signalverkehr	in MODE	Force-load	ONE Shot / Continuous	externer Signalverkehr		Start/Stop

Bild 49. Das Kontrollregister des Timers A.

7	6	5	4	3	2	1	0
ALARM	In MODE		Force-load	ONE Shot/ Continuous	externer Signalverkehr		Start/Stop

Bild 50. Dasselbe für den Timer B.

Register Name	Nr.	7	6	5	4	3	2	1	0
TOD10THS	08	unbenutzt				$\frac{1}{10}$-Sekundenwert			
TODSEC	09	unbenutzt	Zehnerstelle Sekunden			Einerstelle Sekunden			
TODMIN	0A	unbenutzt	Zehnerstelle Minunten			Einerstelle Minuten			
TODHR	0B	AM/PM Flagge	unbenutzt		Zehnerstelle Stunden	Einerstelle Stunden			

Bild 51. Die Register der Echtzeituhren.

gültig, ob der Timer gerade läuft oder nicht. Schreibt man eine Null ein, hat das keine Wirkung.

Beim Lesen des Registers ist dieses Bit immer 0.

Zu diesem Bit und seiner Wirkung ist noch etwas zu sagen. Das Neuladen des Timers geschieht

- immer dann, wenn ein Unterlauf stattgefunden hat oder
- falls der Timer steht und in die Register ein Startwert geschrieben wird. Dabei ist der CIA so konstruiert, daß man kein zwangsweises Laden (also mit Bit 4 = 1) braucht, wenn man den Startwert in der Reihenfolge LSB MSB in die Register bringt.

Die Bits 5 bis 7 haben nun unterschiedliche Bedeutung im CRA und im CRB:

Register CRA ($0E)

Bit 5: Ist dieses Bit gleich Null, dann wird im Systemtakt gezählt. Den hatten wir vorhin zur Zeitberechnung schon verwendet. Wenn das Bit auf 1 gesetzt ist, zählt der Timer externe Signale.

Bit 6: Spielt für den Signalverkehr über den seriellen Port eine Rolle und soll uns hier nicht weiter beschäftigen.

Bit 7: Damit steuert man nicht den Timer A, sondern dieses Bit bezieht sich auf die gleich noch zu behandelnde Echtzeituhr.

Register CRB ($0F)

Die Bits 5 und 6 sind hier im Zusammenhang von Bedeutung. Es gibt vier Kombinationsmöglichkeiten:

Bit 6 - Bit 5	Der Timer B wird - wie vorhin der Timer A - im
0 - 0	Systemtakt heruntergezählt.
0 - 1	Der Timer B wird durch externe Signale heruntergezählt.
1 - 0	Der Timer B zählt die Unterläufe von Timer A. Das ist der vorhin erwähnte Punkt, der beide Timer kombiniert zum 32-Bit-Zähler. Man kann also im Extremfall 65536 mal 65536 Takte zählen lassen.
1 - 1	Auch in diesem Fall zählt Timer B die Unterläufe von Timer A. Er tut das aber nur, wenn ein bestimmtes externes Signal vorhanden ist.

Bit 7: Auch beim Register CRB steuert dieses Bit bestimmte Möglichkeiten der Echtzeituhr. Deshalb haben Sie noch ein wenig Geduld, bis wir diese Uhr behandeln.

Wir kennen uns nun ganz gut aus, wie wir mit den Timern umzugehen haben. Unser Wissen soll in einem kleinen Test erprobt werden. Dazu bedienen wir uns des $\frac{1}{60}$ Sekunden IRQ. Wir verändern diese regelmäßige Unterbrechung derart, daß sie nur noch einmal in der Sekunde geschieht. Welche Zahlen dazu in ein 32-Bit-Register gepackt werden müssen, haben wir schon vorhin berechnet. Jeweils in der Reihenfolge LSB/MSB müssen wir sie einschreiben und vorher die Timer anhalten, indem die Bits 0 der Kontrollregister CRA und CRB auf 0 gesetzt werden. Nach dem Einschreiben und Starten der beiden Timer müssen folgende Bitmuster in CRA und CRB stehen:

CRA

Bit 0 = 1 Start Timer A
Bit 3 = 0 Dauerlauf
Bit 5 = 0 Systemtakt

CRB

Bit 0 = 1 Start Timer B
Bit 3 = 0 Dauerlauf
Bit 5 = 0
Bit 6 = 1 Timer B zählt Unterläufe von Timer A.

Bevor wir die Timer starten, muß auch noch das Interrupt-Kontrollregister verändert werden (das hatten wir uns in Kapitel 52 genauer angesehen). Bislang erzeugt ein Unterlauf des Timers A eine Unterbrechung. Wir möchten aber, daß der Timer B (damit wir das 32-Bit-Register voll ausnutzen) der Auslöser ist. Dazu muß Bit 0 des ICR gelöscht und stattdessen Bit 1 gesetzt werden.

Im Programm »Timer-Test« (siehe Listing 8 und 9) ist all das realisiert. Mit SYS 49152 gestartet, zeigt sich sofort ein deutlich verlangsamter Cursor. Noch langsamer kann alles werden, indem Sie höhere Werte in die Timer-Register schreiben.

```
.programm : prg.timer-testc000 c051
-----------------------------------

c000 : 78 ad 0e dc 29 fe 8d 0e  4b
c008 : dc ad 0f dc 29 fe 8d 0f  f9
c010 : dc a9 0f 8d 06 dc a9 00  24
c018 : 8d 07 dc a9 a0 8d 04 dc  d5
c020 : a9 08 8d 05 dc a9 1f 8d  84
c028 : 0d dc a9 82 8d 0d dc ad  6e
c030 : 0e dc 29 d7 8d 0e dc ad  0a
c038 : 0f dc 29 d7 8d 0f dc ad  1b
c040 : 0e dc 09 01 8d 0e dc ad  37
c048 : 0f dc 09 41 8d 0f dc 58  a5
c050 : 60 ff 00 ff 00 ff 00 ff  b0
```

Listing 8. Programm Timer-Test, ein Beispiel für die Anwendung eines 32-Bit-Timers.

```
PASS 1

PASS 2
7000          0823 ;
7000          084C ;*************************************
7000          0875 ;*                                   *
7000          089E ;*             TIMER-TEST            *
7000          08C7 ;*                                   *
7000          08F0 ;* TIMER A UND B DES CIA1 WERDEN SO  *
7000          0919 ;* GESCHALTET, DASS NUR NOCH 1 MAL   *
7000          0942 ;* PRO SEKUNDE DER TIMER-IRQ AUFTRITT *
7000          096B ;*                                   *
7000          0994 ;*   HEIMO PONNATH  HAMBURG  1985    *
7000          09BD ;*                                   *
7000          09E6 ;*************************************
7000          09E9 ;
7000          09EC ;
C000          09F9             .BA $C000
C000          09FE             .OS
C000          0A01 ;
C000          0A27 ;+++ BENUTZTE ADRESSEN DES CIA 1 +++
C000          0A2A ;
C000          0A3A TALO        .DE $DC04
C000          0A4A TAHI        .DE $DC05
C000          0A5A TBLO        .DE $DC06
C000          0A6A TBHI        .DE $DC07
C000          0A79 ICR         .DE $DC0D
C000          0A88 CRA         .DE $DC0E
C000          0A97 CRB         .DE $DC0F
C000          0A9A ;
C000          0AC3 ;+++ EINSCHALTEN DES 1 SEKUNDEN IRQ +++
C000          0AC6 ;
C000 78       0AE5 START       SEI                 ;SPERREN ALLER IRQS
C001          0AE8 ;
C001 AD 0E DC 0AF2             LDA CRA
C004 29 FE    0B03             AND #%11111110
C006 8D 0E DC 0B1B             STA CRA             ;STOP TIMER A
C009 AD 0F DC 0B25             LDA CRB
C00C 29 FE    0B36             AND #%11111110
C00E 8D 0F DC 0B4E             STA CRB             ;STOP TIMER B
C011          0B51 ;
C011 A9 0F    0B6F             LDA #15             ;NEUER STARTWERT IN
C013 8D 06 DC 0B8A             STA TBLO            ;32-BIT-REGISTER
C016 A9 00    0B94             LDA #00
C018 8D 07 DC 0B9F             STA TBHI
C01B A9 A0    0BAA             LDA #160
C01D 8D 04 DC 0BB5             STA TALO
C020 A9 08    0BBF             LDA #08
C022 8D 05 DC 0BCA             STA TAHI
C025          0BCD ;
C025 A9 1F    0BDE             LDA #%00011111
C027 8D 0D DC 0BFB             STA ICR             ;ALLE IRQ VERBOTEN
C02A A9 82    0C0C             LDA #%10000010
C02C 8D 0D DC 0C27             STA ICR             ;NUR TIMER B IRQ
C02F          0C2A ;
C02F AD 0E DC 0C34             LDA CRA
C032 29 D7    0C45             AND #%11010111
C034 8D 0E DC 0C61             STA CRA             ;BITS 3 UND 5 = 0
C037          0C64 ;
C037 AD 0F DC 0C6E             LDA CRB
C03A 29 D7    0C7F             AND #%11010111
C03C 8D 0F DC 0C8F             STA CRB             ;DITO
C03F          0C92 ;
C03F AD 0E DC 0C9C             LDA CRA
C042 09 01    0CAD             ORA #%00000001
C044 8D 0E DC 0CC6             STA CRA             ;TIMER A START
C047          0CC9 ;
C047 AD 0F DC 0CD3             LDA CRB
C04A 09 41    0CE4             ORA #%01000001
C04C 8D 0F DC 0D01             STA CRB             ;TIMER B START MIT
C04F          0D24 ;               TIMER A UNTERLAUF
C04F          0D27 ;
C04F 58       0D3D             CLI                 ;IRQS FREIGEBEN
C050          0D40 ;
C050 60       0D46             RTS
C051          0D49 ;
C051          0D4F             .EN
```

Listing 9. Der Quelltext zum Timer-Set.

Den Normalzustand stellen Sie einfach durch Drücken der RUN/STOP- und der RESTORE-Tasten her. Dabei wird ja – wie Sie aus dem letzten Kapitel her wissen, auch ein I/O-Reset ausgeführt, der den Ausgangszustand wiederherstellt.

Die Verlängerung des IRQ-Zyklus hat übrigens noch einen sinnvollen Nebeneffekt. Je seltener ein laufendes Programm unterbrochen wird, desto schneller wird es mit seinen Jobs fertig. Das kann man immer dann tun – im Extremfall sogar den IRQ ganz ausschalten – wenn man die Möglichkeiten, die der Computer während des normalen IRQ anbietet, nur selten oder aber gar nicht braucht.

65. Die Echtzeituhren

Wir kennen nun fünf Uhren in unserem Computer: Die vier Timer (jeweils A und B im CIA1 und CIA2), die wir, weil wir die Impulszahlen in Zeiteinheiten umrechnen können, zur Zeitmessung einsetzen könnten und die im Basic verfügbare Uhr TI$, die aber – wie wir nun wissen – lediglich die Umsetzung des Timers A im CIA1 in ein bequemer handhabbares Software-Instrument ist. Zudem ist die Ganggenauigkeit dieser Uhr recht gering. Schon einige Kassettenoperationen genügen, sie völlig aus dem Takt zu bringen.

Um so mehr verwundert es, daß zwei hervorragende Echtzeituhren im Commodore 64 so gut wie nie benutzt werden, ja nicht einmal in irgendeiner Weise softwaremäßig unterstützt werden. Vielleicht ist das ein bißchen zuviel »mehr sein als scheinen«, was Commodore da betreibt, wenn man bedenkt, welche verborgenen Schätze da alle zutage gefördert werden können (man denke nur an die hochauflösende Grafik) bei genauer Untersuchung des Computers.

Jeder der beiden CIAs verfügt über solch eine Uhr, die direkt von der Netzfrequenz getaktet wird. Die Zählung der Zeit geschieht in vier Registern (Register $08 bis $0B), die in Bild 51 gezeigt sind.

Vielleicht fällt Ihnen etwas auf, wenn Sie sich diese vier Byte mal genauer ansehen: Die Speicherung geschieht in Form von Einer- und Zehnerstellen. Das kann also weder im Binärformat noch als ASCII-Zeichen funktionieren. Hier werden die Ziffern als BCD-Zahlen abgelegt. In Kapitel 13 wurde dieses »binary coded decimal«-Format erklärt. Das ist lange her und soll deshalb hier nochmal vorgestellt werden, damit alle wissen, wovon die Rede ist.

In dieser Zahlendarstellung wird jede Dezimalstelle einer Zahl gesondert in eine Binärzahl umgewandelt. Dann ergibt sich der folgende Zusammenhang:

Binär	Dezimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9

Das war's! Die anderen möglichen Binärkombinationen (also zum Beispiel 1010 etc.) werden nicht benutzt. Die Zahl 25 beispielsweise lautet im BCD-Format:

```
0010  0101
  ↑     ↑
  2     5
```

Jetzt ist es Ihnen sicherlich verständlich, warum für die Sekunden- und Minuten-Zehnerstellen nicht mehr als drei Bit reserviert wurden: größer als 6 wird die Zehnerstelle nicht.

Zum Stundenregister TODHR ist aber noch etwas zu sagen: Dort ist nur ein Bit reserviert für die Stunden-Zehnerstelle. Die Uhr läuft nicht bis 24 Uhr, sondern lediglich bis 12 Uhr. Zur Unterscheidung, ob vor- oder nachmittags gemeint ist, dient das Bit 7. Dieses sogenannte AM/PM-Flag ist orientiert an der angelsächsischen Gewohnheit, zum Beispiel für 16 Uhr den Ausdruck 4 PM zu verwenden. PM kommt vom lateinischen »post meridiem«, was übersetzt heißt »nach dem Mittag«, wohingegen AM steht für »ante meridiem«, also »vor dem Mittag«. Meint man nun AM, dann muß diese Flagge auf 0, bei PM aber auf 1 gesetzt sein.

Beim Stellen der Uhren sollte eine Reihenfolge eingehalten werden. Sobald nämlich in das Stundenregister geschrieben wird, hält die Zählung automatisch an. Man kann nun die anderen Werte in die Register schreiben. Den Startschuß liefert das Schreiben in das Register TOD10TH: von nun an tickt die Uhr wieder.

Ähnlich funktioniert das Lesen der Uhrzeit. Sobald das Stundenregister gelesen wird, führt das zum Anhalten der Uhr, so daß die restlichen Register reibungslos auslesbar sind. Wieder ist es das Zehntelsekundenregister, das beim Auslesen ein Weiterlaufen der Uhr bewirkt. Aber, so werden Sie bemerken, wenn der Auslesevorgang eine bestimmte Zeit beansprucht, führt das zu Verzögerungen? Die Lösung ist, daß der gesamte Inhalt der vier Register gleichzeitig mit dem Auslesen des Stundenwertes in einen internen Speicher transferiert wird und dort weiterläuft. Nach dem Lesen des TOD10TH kommt der aktuelle Wert zurück in die Register und dieser wird weitergezählt.

Nun wird es höchste Zeit, daß wir uns die beiden Bits im CRA und im CRB ansehen, die wir vorhin bei der Timer-Behandlung links liegen ließen. Bit 7 im CRA kündigt der Echtzeituhr an, welche Netzfrequenz zu erwarten ist. Eine 1 an dieser Stelle steht für 50 Hz, eine 0 für 60 Hz. Unser Stromnetz in Deutschland liefert einen Wechselstrom mit 50 Hz, weshalb wir dann dort die 1 setzen sollten. Da gibt es ein kleines Problem: Beim I/O-Reset, der durch Drücken der RUN/STOP- und der RESTORE-Tasten zusammen ausgelöst wird, schreibt der Computer immer den amerikanischen Wert für 60 Hz in dieses Bit. Dann geht die Uhr aber empfindlich nach. Man muß also einen Weg finden, der erlaubt, dort in diesem Fall wieder eine 1 einzuschreiben. Das ist durch eine eigene NMI-Routine möglich. Sie sehen schon, der Weg zur Nutzung dieser verlockenden Uhren ist ziemlich dornenreich!

Noch interessanter ist das Bit 7 im CRB. Das Setzen der Uhrzeit ist nämlich nur möglich, wenn dieses Bit den Inhalt 0 hat. Was geschieht, wenn dort eine 1 steht? Dann bestimmt man nicht die aktuelle Uhrzeit, sondern man stellt einen Wecker (das ist die Alarmzeit). Das geschieht nach dem Setzen dieses Bits genauso wie vorhin das Einschreiben der Uhrzeit (also erstaunlicherweise auch in genau dieselben Register!). Im Unterschied dazu ist allerdings ein Lesen der Alarmzeit nicht möglich – das ergibt immer die aktuelle Uhrzeit. Man muß für diesen Fall die Weckzeit irgendwo abspeichern und bei Bedarf dann von dort lesen.

Weil man ja meistens nach dem Erreichen der Alarmzeit irgendeine Reaktion erwartet, ist im ICR (also dem Unterbrechungskontrollregister 13) jedes CIAs noch ein Bit reserviert – das Bit 2 —, mit dessen Hilfe der Alarm per IRQ oder NMI wie auch immer geartet losbrechen kann. Der Phantasie sind hier nur wenige Grenzen gesetzt. Wie man mit diesem ICR umgeht, haben wir schon besprochen.

Damit sind wir durch die Eigenheiten der CIAs durch. Man braucht tatsächlich keine Scheu zu haben, diese Echtzeituhren zu nutzen. Lediglich die Uhr im CIA1 wird manchmal verwendet, einen bestimmten Wert für die Zufallszahlenerzeugung zu generieren. Aber das sollte einer eigenen Uhren-

Routine nicht in die Quere kommen. Solch eine Echtzeituhr finden Sie in Listing 10 und 11.

Durch SYS49152 aktivieren Sie die Uhr, die Sie mit SYS49261 auch wieder abschalten können. Durch ein USR-Kommando A=USR (String) stellen Sie die Startzeit ein. String kann dabei eine Stringvariable sein oder auch direkt ein String der Form »HHMMSST« (also Stunden, Minuten, Sekunden, Zehntelsekunden). In A steht eine 0, wenn kein Fehler, aber eine -1, wenn ein Fehler aufgetreten ist. Das Lesen der Uhr erfolgt über ein zweites USR-Kommando:

PRINTUSR(Zahl). Dabei kann Zahl eine beliebige Zahl oder Variable sein. Eine Alarmzeit ist ebenfalls einstellbar durch ein USR-Kommando, in dem vor der Zeiteingabe noch ein Buchstabe steht. Beispielsweise stellt A=USR(»A1200000«) einen Wecker auf 12 Uhr. Der Alarm im Programm läßt den Bildschirmrahmen blinken. Abstellen kann man das durch Auslösen eines RESTORE-NMI (also RUN/STOP und RESTORE). Sollten Sie vor dem eingestellten Alarm mal solch einen NMI auslösen, dann muß die Alarmzeit neu gestellt werden. Als Basis für dieses Programm diente ein Listing aus dem schon oft erwähnten Buch von Babel/Krause/Dripke »Das Interface Age Systemhandbuch zum Commodore 64«.

Die Unterbrechungs-Programmierung ist damit abgeschlossen – ebenso dieser Kurs, der als Einführung in die Assembler-Alchimie nun alle Geheimnisse der Kunst aufgedeckt hat. In den letzten Kapiteln sind wir schon in die Meistergrade der Zunft aufgestiegen. Vielleicht ging es manchem etwas zu schnell? Dann wird Ihnen der Kurs »Von Basic zu Assembler« eine Hilfe sein, der behutsam und mit vielen an Basic angelehnten Beispielen die nötige Programmierpraxis vermitteln wird (ab 64'er, Ausgabe 1/86). So wie die Segler sich oft »Mast- und Schotbruch« wünschen, verabschiede ich mich, indem ich Ihnen viele grandiose Abstürze wünsche.

(Heimo Ponnath/gk)

```
programm : obj.alarmuhr c000 c18d
----------------------------------

c000 : a9 8e 8d 11 03 a9 c0 8d  11
c008 : 12 03 a9 1d 8d 18 03 a9  a3
c010 : c0 8d 19 03 ad 0e dd 09  12
c018 : 80 8d 0e dd 60 48 8a 48  a1
c020 : 98 48 a9 7f 8d 0d dd ac  49
c028 : 0d dd 10 06 4c 6a c1 4c  a0
c030 : 72 fe 20 bc f6 20 e1 ff  b9
c038 : d0 f5 a2 04 bd 2f fd 9d  b4
c040 : 13 03 ca d0 f7 a2 1a bd  1a
c048 : 35 fd 9d 19 03 ca d0 f7  c0
c050 : a9 7f 8d 0d dc 8d 0d dd  e8
c058 : 8d 00 dc a9 08 8d 0e dc  30
c060 : a9 88 8d 0e dd a9 08 20  fe
c068 : b6 fd 4c 6c fe a9 48 8d  37
c070 : 11 03 a9 b2 8d 12 03 78  2a
c078 : a9 47 8d 18 03 a9 fe 8d  c0
c080 : 19 03 a9 31 8d 14 03 a9  84
c088 : ea 8d 15 03 58 60 24 0d  12
c090 : 30 03 4c 20 c1 20 82 b7  f0
c098 : c0 07 d0 40 ad 0f dd 29  35
c0a0 : 7f 8d 0f dd a0 00 a9 24  5e
c0a8 : 20 fe c0 d0 02 a9 24 c9  23
c0b0 : 13 90 07 f8 38 e9 12 d8  b9
c0b8 : 09 80 8d 0b dd 20 fc c0  1a
c0c0 : 8d 0a dd 20 fc c0 8d 09  ec
c0c8 : dd 20 66 c1 8d 08 dd a9  6b
c0d0 : 00 4c 3c bc 68 68 68 68  d9
c0d8 : a9 ff d0 f5 c0 08 d0 f8  f5
c0e0 : ad 0f dd 09 80 8d 0f dd  1a
c0e8 : a9 84 8d 0d dd a9 3c 85  ff
c0f0 : 04 85 02 a9 ff 85 03 a0  e6
c0f8 : 01 4c a6 c0 a9 60 85 24  dd
c100 : 20 13 c1 0a 0a 0a 0a 85  80
c108 : 25 20 13 c1 05 25 c5 24  13
c110 : b0 c4 60 b1 22 38 e9 30  5d
c118 : 90 ba c9 0a b0 b6 c8 60  5e
c120 : a9 07 20 7d b4 a0 00 ad  b0
c128 : 0b dd 08 29 1f c9 12 d0  73
c130 : 02 a9 00 28 10 05 f8 18  49
c138 : 69 12 d8 20 55 c1 ad 0a  13
c140 : dd 20 55 c1 ad 09 dd 20  96
c148 : 55 c1 ad 08 dd 20 60 c1  ce
c150 : 68 68 4c ca b4 48 4a 4a  a4
c158 : 4a 4a 20 60 c1 68 29 0f  fe
c160 : 09 30 91 62 c8 60 20 13  68
c168 : c1 60 a9 77 8d 14 03 a9  8b
c170 : c1 8d 15 03 4c bc fe c6  d2
c178 : 02 f0 03 4c 31 ea a5 04  46
c180 : 85 02 ad 20 d0 45 03 8d  d4
c188 : 20 d0 4c 31 ea 00 ff 00  f8
```

Listing 10. Eine Echtzeituhr.

```
PASS 2
7000               0823 ;
7000               084C ;**************************************
7000               0875 ;*                                    *
7000               089E ;*  ECHTZEITUHR MIT ALARMFUNKTION     *
7000               08C7 ;*                                    *
7000               08F0 ;*  LAEUFT MIT DEM NMI-CIA            *
7000               0919 ;*  IN VERBINDUNG MIT DEM IRQ FUER    *
7000               0942 ;*  DEN ALARM                         *
7000               096B ;*                                    *
7000               0994 ;*    HEIMO PONNATH  HAMBURG  1985    *
7000               09BD ;*                                    *
7000               09E6 ;*  (TEILWEISE WURDE EIN PROGRAMM AUS *
7000               0A0F ;*  DEM INTERFACE AGE SYSTEMHANDBUCH  *
7000               0A38 ;*  ZUM COMMODORE 64 , SEITE 114      *
7000               0A61 ;*  ALS BASIS VERWENDET )             *
7000               0A8A ;*                                    *
7000               0AB3 ;**************************************
7000               0AB6 ;
C000               0AC2                .BA $C000
C000               0AC8                .OS
C000               0ACB ;
C000               0AF4 ;*********** ZEROPAGE-LABELS ***********
C000               0AF7 ;
C000               0B18 VERZ           .DE $02              ;AKTUELLE VERZOEG.
C000               0B38 FARB           .DE $03              ;WERT FUER RAHMEN
C000               0B57 ;                  EOR-OPERATION
C000               0B78 VORW           .DE $04              ;VERZOEGERUNGSWERT
C000               0B9B VALTYP         .DE $0D              ;INHALT:FF=STR 0=N
C000               0B9E ;
C000               0BAD INDEX          .DE $22
C000               0BC6 INDEX3         .DE $24              ;POINTER
C000               0BD6 INDEX4         .DE $25
C000               0BF5 FAC1           .DE $62              ;1.MANTISSENBYTE
C000               0BF8 ;
C000               0C21 ;*********** LABELS PAGES 3 ***********
C000               0C24 ;
C000               0C44 USRADDL        .DE $0311            ;USR-POINTER
C000               0C57 USRADDH        .DE $0312
C000               0C5A ;
C000               0C6A FREI           .DE $0313
C000               0C6D ;
C000               0C8A IRQVL          .DE $0314            ;IRQ-VEKTOR
C000               0C9B IRQVH          .DE $0315
C000               0C9E ;
C000               0CBC NMINVL         .DE $0318            ;NMI-VEKTOR
C000               0CCE NMINVH         .DE $0319
C000               0CD1 ;
C000               0CFA ;*********** LABELS INTERPRETER ********
C000               0CFD ;
C000               0D23 ILLQUERR       .DE $B248            ;ILLEGAL QUANTITY
C000               0D4B ;             ERROR=NORMALWERT USR-VEKTOR
C000               0D4E ;
C000               0D70 STRINI8        .DE $B47D            ;SPEICHERPLATZ
C000               0D99 ;             PRUEFEN,STRINGPOINTER SETZEN
C000               0DBF STRLIT67       .DE $B4CA            ;REST DER STRING-
C000               0DD8 ;             LESE-ROUTINE
C000               0DF7 LEN1           .DE $B782            ;STRINGLAENGE
C000               0E11 ;              IN Y-REGISTER
C000               0E32 ACTOFC         .DE $BC3C            ;AKKU NACH FAC
C000               0E35 ;
C000               0E5E ;*********** LABELS VIC-II-CHIP ********
C000               0E61 ;
C000               0E7E RAND           .DE $D020            ;RAHMENFARBE
C000               0E81 ;
C000               0EAA ;*********** LABELS CIA-BAUSTEINE ******
C000               0EAD ;
C000               0ECA CIA1           .DE $DC00            ;START CIA-1
C000               0EEC ICR1           .DE $DC0D            ;IRQ-KONTROLLREG.
C000               0F0E CRA1           .DE $DC0E            ;TIMER-A KONTRREG
C000               0F11 ;
C000               0F34 TOD10TH2       .DE $DD08            ;1/10 SEKUNDEN
C000               0F51 TODSEC2        .DE $DD09            ;SEKUNDEN
C000               0F6D TODMIN2        .DE $DD0A            ;MINUTEN
C000               0F8F TODHR          .DE $DD0B            ;STUNDEN + AM/PM
C000               0FB1 ICR2           .DE $DD0D            ;NMI-KONTROLLREG.
C000               0FD3 CRA2           .DE $DD0E            ;TIMER-A KONTRREG
C000               0FF5 CRB2           .DE $DD0F            ;TIMER-B KONTRREG
C000               0FF8 ;
C000               0FFB ;
C000               1024 ;********** LABELS OBERES ROM **********
C000               1027 ;
C000               1045 NORM           .DE $EA31            ;NORMALER IRQ
C000               1048 ;
C000               1068 TASTFLAG       .DE $F6BC            ;TEIL DER NMI-
C000               108E ;               ROUTINE (KEIN MODUL)
C000               10AD VECTAB         .DE $FD2F            ;TABELLE DER
C000               10C8 ;               ROM-VEKTOREN
C000               10E9 VECTAB7        .DE $FD35            ;MSB DES NMI-
C000               110E ;               VEKTORS IN DER TABELLE
C000               1132 IORESET19      .DE $FDB6            ;I/O-RESET:BEI
C000               1157 ;               SETZEN DES CRA IRQ-CIA
C000               117B NMIXCT16       .DE $FE6C            ;NMI-ROUTINE AB
C000               119D ;               SCREEN-EDITOR-RESET
C000               11C1 NMIRS232       .DE $FE72            ;NMI-ROUTINE AB
C000               11DE ;               RS232-HANDLING
C000               11FF NMIEND         .DE $FEBC            ;ENDE DER NMI-
C000               1215 ;               ROUTINE
C000               1237 STOP           .DE $FFE1            ;KERNAL STOP SPRG
C000               1254 ;               NACH JMP($328)
C000               1257 ;
C000               125A ;
C000               1283 ;*********** AKTIVIEREN ****************
C000               1286 ;
C000 A9 8E         12A2 INIT           LDA #L,USR           ;USR-VEKTOR
C002 8D 11 03      12B7                STA USRADDL          ;LADEN
C005 A9 C0         12C4                LDA #H,USR
C007 8D 12 03      12D2                STA USRADDH
C00A               12D5 ;
C00A A9 1D         12F2                LDA #L,NMI           ;NMI-VEKTOR MIT
C00C 8D 18 03      130D                STA NMINVL           ;STARTADRESSE
C00F A9 C0         1327                LDA #H,NMI           ;DER EIGENEN
C011 8D 19 03      1345                STA NMINVH           ;NMI-ROUTINE LAD
C014               1348 ;
C014 AD 0E DD      1365                LDA CRA2             ;BIT7 CRA SETZEN:
C017 09 80         137C                ORA #$80             ;%1000 0000
C019 8D 0E DD      1397                STA CRA2             ;NETZFREQ.=50HZ
C01C               139A ;
C01C 60            13A0                RTS
C01D               13A3 ;
C01D               13CC ;*********** EIGENE NMI-ROUTINE ********
C01D               13CF ;
C01D 48            13EF NMI            PHA                  ;ANFANG NORMALE NMI-R.
```

Listing 11. Der Quelltext zur Echtzeituhr.

```
C01E 8A       1406            TXA                  ;REGISTER RETTEN
C01F 48       140C            PHA
C020 98       1412            TYA
C021 48       1418            PHA
C022          141B ;
C022 A9 7F    1439            LDA #$7F             ;SPERREN ALLER NMI
C024 8D 0D DD 1444            STA ICR2
C027          1447 ;
C027 AC 0D DD 1462            LDY ICR2             ;PRUEFEN OB NMI
C02A 10 06    1481            BPL RESTNMI          ;VOM CIA2 KOMMT.
C02C          14A4 ;              WENN NEIN=SPRUNG
C02C 4C 6A C1 14C1            JMP ALARM            ;WENN JA, ALARM
C02F 4C 72 FE 14E1 CIANMI     JMP NMIRS232         ;REST DER
C032          1508 ;              NORMALEN NMI-ROUTINE
C032          150B ;
C032          1534 ;***** EIGENE RESTORE-NMI-ROUTINE *****
C032          1537 ;
C032          155F ;   DIE MODULPRUEFUNG WIRD AUSGELASSEN
C032          1562 ;
C032 20 BC F6 1587 RESTNMI    JSR TASTFLAG         ;TEIL DER NMI-
C035 20 E1 FF 15A5            JSR STOP             ;ROUTINE ZUR STOP-
C038 D0 F5    15C2            BNE CIANMI           ;TASTEN-ABFRAGE
C03A          15C5 ;
C03A A2 04    15E3            LDX #$04             ;IRQ UND BRK VEKT.
C03C BD 2F FD 1606 UMLAD1     LDA VECTAB,X         ;RESTAURIEREN
C03F 9D 13 03 1613            STA FREI,X
C042 CA       1619            DEX
C043 D0 F7    1626            BNE UMLAD1
C045          1629 ;
C045          1651 ;   DER NMI-VEKTOR WIRD UEBERSPRUNGEN
C045          1654 ;
C045 A2 1A    1671            LDX #$1A             ;RESTAURIEREN DER
C047 BD 35 FD 1693 UMLAD2     LDA VECTAB7,X        ;RESTLICHEN
C04A 9D 19 03 16AC            STA NMINVH,X         ;VEKTOREN
C04D CA       16B2            DEX
C04E D0 F7    16BF            BNE UMLAD2
C050          16C2 ;
C050          16E3 ;  ZUNAECHST NORMALER I/O-RESET
C050          16E6 ;
C050 A9 7F    16FD            LDA #$7F             ;= 0111 111
C052 8D 0D DC 171B            STA ICR1             ;SPERREN ALLER IRQ
C055 8D 0D DD 1739            STA ICR2             ;SPERREN ALLER NMI
C058 8D 00 DC 1753            STA CIA1             ;DATENREGISTER
C05B          177A ;              PORT A AUF NORMALWERT
C05B A9 08    1791            LDA #$08             ;=0000 1000
C05D 8D 0E DC 17AD            STA CRA1             ;TIMER A IM CIA1
C060          17B0 ;
C060          17D2 ;  ERSATZ FUER BELEGUNG DES CRA2
C060          17D5 ;
C060 A9 88    17EC            LDA #$88             ;=1000 1000
C062 8D 0E DD 1809            STA CRA2             ;TIMER A IM CIA2:
C065          1829 ;              BIT 0 AUF STOP
C065          184F ;              BIT 3 AUF EINZELLAUF
C065          1875 ;              BIT 5 SYSTEMTAKT EIN
C065          189D ;              BIT 7 ECHTZEITUHR=50HZ
C065          18A0 ;
C065          18C0 ;  REST DES NORMALEN I/O-RESET
C065          18C3 ;
C065          18C6 ;
C065 A9 08    18DE            LDA #$08             ;= 0000 1000
C067 20 B6 FD 18EE            JSR IORESET19
C06A          18F1 ;
C06A          1919 ;  REST DER NORMALEN RESTORE-NMI-ROUT.
C06A          191C ;
C06A 4C 6C FE 193A            JMP NMIXCT16         ;EINSPRUNG BEI
C06D          195F ;              SCREEN EDITOR RESET
C06D          1962 ;
C06D          198B ;**** ABSCHALTEN DER TIME OF DAY UHR **
C06D          199E ;
C06D          19AD ;          DURCH SYS-KOMMANDO
C06D          19B0 ;
C06D A9 48    19D1 AUS        LDA #L,ILLQUERR      ;USR-VEKTOR
C06F 8D 11 03 19EF            STA USRADDL          ;AUF NORMALWERT
C072 A9 B2    1A01            LDA #H,ILLQUERR
C074 8D 12 03 1A0F            STA USRADDH
C077          1A12 ;
C077 78       1A18            SEI
C078 A9 47    1A35            LDA #$47             ;RESTAURIEREN DES
C07A 8D 18 03 1A4F            STA NMINVL           ;NMI-VEKTORS
C07D A9 FE    1A5A            LDA #$FE
C07F 8D 19 03 1A67            STA NMINVH
C082          1A6A ;
C082 A9 31    1A87            LDA #L,NORM          ;RESTAURIEREN
C084 8D 14 03 1AA4            STA IRQVL            ;DES IRQ-VEKTORS
C087 A9 EA    1AB2            LDA #H,NORM
C089 8D 15 03 1ABE            STA IRQVH
C08C          1AC1 ;
C08C 58       1AC7            CLI
C08D 60       1ACD            RTS
C08E          1AD0 ;
C08E          1AF9 ;**** DURCH USR AUFRUFBARE ROUTINE ****
C08E          1AFC ;
C08E 24 0D    1B1D USR        BIT VALTYP           ;WELCHER TYP VON
C090          1B44 ;              VARIABLEN LIEGT VOR ?
C090 30 03    1B5F            BMI STRING           ;WENN STRING,
C092          1B83 ;              DANN UEBERSPRINGEN
C092 4C 20 C1 1B9F            JMP ZAHLVAR          ;SONST SPRUNG
C095          1BA2 ;
C095          1BCB ;****** STELLEN DER ECHTZEITUHR *******
C095          1BEB ;          DURCH USR("HHMMSST")
C095          1BEE ;
C095 20 82 B7 1C0F STRING     JSR LEN1             ;Y=STRINGLAENGE
C098 C0 07    1C2C            CPY #$07             ;STRING=7ZEICHEN?
C09A D0 40    1C4A            BNE ALSET            ;NEIN DANN ALARM
C09C          1C69 ;              ZEIT STELLEN?
C09C          1C6C ;
C09C AD 0F DD 1C88            LDA CRB2             ;TIMER B IN CIA2
C09F 29 7F    1CA3            AND #$7F             ;BIT7 LOESCHEN:
C0A1 8D 0F DD 1CBF            STA CRB2             ;NORMALE UHRZEIT
C0A4          1CE4 ;              IN ECHTZEITUHR CIA2
C0A4          1CE7 ;
C0A4          1D05 ;   AUSLESEN DES ZEIT-STRINGS
C0A4          1D08 ;
C0A4 A0 00    1D22            LDY #$00             ;ZAEHLER AUF 0
C0A6 A9 24    1D47 STELLEN    LDA #$24             ;BCD 24 STD-VERGL.
C0A8 20 FE C0 1D65            JSR ASCBCD           ;ZEICHENTEST UND
C0AB          1D8D ;              UMWANDLUNG IN BCD-ZAHL
C0AB D0 02    1DAB            BNE STD12            ;STUNDEN UNGLEICH
C0AD          1DCF ;              NULL ? DANN SPRUNG
C0AD A9 24    1DE6            LDA #$24             ;SONST = 24
C0AF          1DE9 ;
C0AF C9 13    1E0B STD12      CMP #$13             ;STUNDEN GROESSER
C0B1 90 07    1E2A            BCC STDSET           ;ODER GLEICH 12 ?
```

```
C0B3          1E4D ;              NEIN, DANN SPRUNG
C0B3 F8       1E67            SED                  ;SONST DAVON BCD 12
C0B4 38       1E7B            SEC                  ;SUBTRAHIEREN
C0B5 E9 12    1E8B            SBC #$12             ;UND
C0B7 D8       1E91            CLD
C0B8 09 80    1EA9            ORA #$80             ;BIT7 SETZEN
C0BA          1EAC ;
C0BA 8D 0B DD 1ECF STDSET     STA TODHR            ;BCD-STUNDEN UND
C0BD          1EF7 ;              AM/PM-FLAG IN TOD-CIA2
C0BD          1EFA ;
C0BD 20 FC C0 1F18            JSR ASCBCD1          ;ZEICHENTEST U.
C0C0          1F3F ;              UMWANDELN IN BCD-ZAHL
C0C0 8D 0A DD 1F5A            STA TODMIN2          ;ERGEBNIS IN
C0C3          1F7F ;              TOD-MINUTENREGISTER
C0C3          1F82 ;
C0C3 20 FC C0 1F9F            JSR ASCBCD1          ;DASSELBE FUER
C0C6 8D 09 DD 1FBB            STA TODSEC2          ;DIE SEKUNDEN
C0C9          1FBE ;
C0C9 20 66 C1 1FDA            JSR TEST             ;PRUEFEN,OB 1/10
C0CC          1FF9 ;              SEKUNDEN=ZAHL
C0CC 8D 08 DD 2017            STA TOD10TH2         ;UND EINTRAGEN
C0CF          2039 ;              INS TOD-REGISTER
C0CF          205D ;  DIE UHR BEGINNT JETZT ZU LAUFEN
C0CF          2060 ;
C0CF A9 00    207D            LDA #$00             ;KENNUNG FUER OK.
C0D1 4C 3C BC 20A2 AKKUFAC    JMP ACTOFC           ;AKKU ZUR UEBER-
C0D4          20C9 ;              GABE INS BASIC IN FAC
C0D4          20CC ;
C0D4          20F5 ;*********** FEHLER AUFGETRETEN ********
C0D4          20F8 ;
C0D4 68       2116 FEHLER     PLA                  ;JSR-ADRESSEN VOM
C0D5 68       212A            PLA                  ;STAPEL HOLEN
C0D6          212D ;
C0D6 68       2138 ERROR      PLA
C0D7 68       213E            PLA
C0D8          2141 ;
C0D8 A9 FF    2164 ERROR1     LDA #$FF             ;FEHLERKENNUNG IN
C0DA D0 F5    2180            BNE AKKUFAC          ;AKKU UND FAC
C0DC          2183 ;
C0DC          21AC ;ENDE DIESES TEILS D. UNBEDINGTEN SPRG.
C0DC          21D5 ;---------------------------------------
C0DC          21D8 ;
C0DC          2201 ;**** ALARMZEIT EINLESEN **************
C0DC          2204 ;
C0DC          222A ;  AUFRUF DURCH Z.B. USR("AHHMMSST")
C0DC          222D ;
C0DC C0 08    224B ALSET      CPY #$08             ; 8 ZEICHEN ?
C0DE D0 F8    2265            BNE ERROR1           ;NEIN=FEHLER
C0E0          2268 ;
C0E0 AD 0F DD 2273            LDA CRB2
C0E3 09 80    228E            ORA #%10000000       ;ALARMBIT
C0E5 8D 0F DD 22A1            STA CRB2             ;SETZEN
C0E8          22A4 ;
C0E8 A9 84    22C0            LDA #%10000100       ;ALARM-NMI
C0EA 8D 0D DD 22D5            STA ICR2             ;ZULASSEN
C0ED          22D8 ;
C0ED A9 3C    22F3            LDA #$3C             ;VERZOEGERUNGS-
C0EF 85 04    230D            STA VORW             ;WERT VORGEBEN
C0F1 85 02    2318            STA VERZ
C0F3 A9 FF    2336            LDA #$FF             ;EOR-WERT VORGEBEN
C0F5 [illegible] 03 2341      STA FARB
C0F7 [illegible] 01 235F      LDY #$01             ;BUCHSTABE UEBERL.
C0F9 4C A6 C0 236D            JMP STELLEN
C0FC          2370 ;
C0FC          2373 ;
C0FC          239C ;****************************************
C0FC          23C5 ;UNTERPROGRAMM ZUR UMWANDLUNG DER ASCII
C0FC          23EC ;CODES IN BCD-ZAHLEN UND PRUEFUNG DER
C0FC          23FF ;EINGABE-ZEICHEN.
C0FC          2402 ;
C0FC A9 60    2427 ASCBCD1    LDA #$60             ;BCD 60 ALS GRENZE
C0FE          244F ;              FUER MIN UND SEK WERTE
C0FE          2452 ;
C0FE 85 24    2465 ASCBCD     STA INDEX3
C100 20 13 C1 2482            JSR TEST1            ;PRUEFEN OB ZAHL
C103 0A       249A            ASL                  ;AUS LSB INS MSB
C104 0A       24AB            ASL                  ;SCHIEBEN
C105 0A       24B2            ASL
C106 0A       24B9            ASL
C107 85 25    24D7            STA INDEX4           ;UND ZW.SPEICHER
C109          24DA ;
C109 20 13 C1 24F7            JSR TEST1            ;NAECHSTE ZIFFER
C10C          2510 ;              PRUEFEN
C10C 05 25    252C            ORA INDEX4           ;MSB AUS ZWSP.
C10E          2552 ;              UND LSB ZUSAMMENOREN
C10E C5 24    256F            CMP INDEX3           ;UNTER GRENZW.?
C110 B0 C4    258D            BCS ERROR            ;NEIN=FEHLERAUSG.
C112          2590 ;
C112 60       2596            RTS
C113          2599 ;
C113          25C2 ;** PRUEFUNG OB ASCII-ZAHL VORLIEGT ***
C113          25C5 ;
C113 B1 22    25E8 TEST1      LDA (INDEX),Y        ;ZEICHEN EIN-
C115          2607 ;              LESEN IN AKKU
C115 38       260D            SEC
C116 E9 30    2625            SBC #$30             ;< ASCII 0 ?
C118 90 BA    263D            BCC FEHLER           ;JA=FEHLER
C11A          2640 ;
C11A C9 0A    2659            CMP #$0A             ;>= ASCII : ?
C11C B0 B6    2671            BCS FEHLER           ;JA=FEHLER
C11E          2674 ;
C11E C8       2690            INY                  ;SCHLEIFENZAEHLER + 1
C11F 60       2696            RTS
C120          2699 ;
C120          26C2 ;**** ENDE PROGRAMMTEIL UHR STELLEN ***
C120          26C5 ;
C120          26EE ;---------------------------------------
C120          26F1 ;
C120          271A ;********** UHR LESEN ****************
C120          271D ;
C120          273E ;      GESCHIEHT DURCH USR(ZAHL)
C120          2741 ;
C120 A9 07    2761 ZAHLVAR    LDA #$07             ;STRINGLAENGE
C122 20 7D B4 277F            JSR STRINIB          ;SCHAFFT 7 BYTE
C125          27A7 ;              PLATZ FUER STRING UND
C125          27CF ;              LEGT START NACH $62/63
C125          27F6 ;              SOWIE LAENGE NACH $61
C125          2814 ;              (FAC $61-66)
C125          2817 ;
C125 A0 00    2831            LDY #$00             ;ZAEHLER AUF 0
C127 AD 0B DD 284F            LDA TODHR            ;STUNDE AUSLESEN,
C12A          2878 ;              DABEI WIRD GESAMTE ZEIT
C12A          28A1 ;              ZWISCHENGESPEICHERT UND
C12A          28C7 ;              ERST NACH LESEN DER
C12A          28EF ;              1/10-SEK ZURUECKGEHOLT
C12A          2914 ;              MIT AKTUELLEN WERT.
```

```
C12A          2917 ;
C12A 08       2935              PHP                 ;STATUS ZWISCHENSPEICH.
C12B          2938 ;
C12B 29 1F    294F              AND #$1F            ;=0001 1111
C12D          2978 ;                LOESCHEN DER AM/PM-FLAG
C12D C9 12    2996              CMP #$12            ;=0001 0010 =BCD12
C12F D0 02    29B0              BNE NO12            ;<>DANN SPRUNG
C131 A9 00    29CE              LDA #$00            ;SONST STATTDESSEN
C133          29D1 ;
C133 28       29F0 NO12         PLP                 ;STATUS ZURUECKHOLEN
C134 10 05    2A0A              BPL AM              ;FALLS KEINE AM/
C136          2A2F ;                PM-FLAG GESETZT WAR
C136          2A32 ;
C136 F8       2A4C              SED                 ;SONST ADDIEREN VON
C137 18       2A62              CLC                 ;BCD 12 WEIL PM
C138 69 12    2A6D              ADC #$12
C13A D8       2A73              CLD
C13B          2A76 ;
C13B 20 55 C1 2A96 AM           JSR BCDASC          ;UP ZUR UMRECHNG
C13E          2ABC ;                VOM BCD IN ASCII UND
C13E          2AE5 ;                ABLEGEN IM STRING. HIER
C13E          2B02 ;                STUNDENWERT
C13E          2B05 ;
C13E AD 0A DD 2B22              LDA TODMIN2         ;DASSELBE FUER
C141 20 55 C1 2B3C              JSR BCDASC          ;MINUTENWERT
C144          2B3F ;
C144 AD 09 DD 2B5C              LDA TODSEC2         ;UND SEKUNDEN-
C147 20 55 C1 2B6F              JSR BCDASC          ;WERT
C14A          2B72 ;
C14A AD 08 DD 2B9C              LDA TOD10TH2        ;UND 1/10-
C14D 20 60 C1 2BA8              JSR BCDASC1         ;SEKUNDENWERT
C150          2BAB ;
C150 68       2BC7              PLA                 ;USR-STRING-ARGUMENT-
C151 68       2BE5              PLA                 ;RUECKSPRUNG VORBEREIT.
C152          2BE8 ;
C152 4C CA B4 2C06              JMP STRLIT67        ;BRINGT STRING
C155          2C29 ;                DESCRIPTOR IN DIE
C155          2C52 ;                DESCRIPTORTABELLE ($19-
C155          2C79 ;                $21),SETZT POINTER IN
C155          2CA2 ;                FAC(HIER $64/65) DARAUF
C155          2CC8 ;                SETZT STRING-FLAGGE,
C155          2CE9 ;                ERHOEHT LETZTEN
C155          2D10 ;                DESCRIPTOR-INDEX UM 3
C155          2D38 ;                ROUTINE ENDET MIT RTS.
C155          2D3B ;
C155          2D64 ;**** ENDE DES LESENS DER UHR *********
C155          2D67 ;
C155          2D90 ;--------------------------------------
C155          2D93 ;
C155          2DBC ;* UNTERPROGRAMM Z. UMRECHNUNG BCD IN *
C155          2DE5 ;* ASCII U.EINTRAGEN IN STRINGSPEICHER*
C155          2DE8 ;
C155 48       2E0B BCDASC       PHA                 ;AUF STAPEL ZW.SPEICH.
C156          2E0E ;
C156 4A       2E2A              LSR                 ;MSB INS LSB SCHIEBEN
C157 4A       2E31              LSR
C158 4A       2E38              LSR
C159 4A       2E3F              LSR
C15A          2E42 ;
C15A 20 60 C1 2E5E              JSR BCDASC1         ;IN ASCII UM-
C15D          2E85 ;                RECHNEN UND SPEICHERN
C15D          2E88 ;
C15D 68       2EA5              PLA                 ;ZURUECKHOLEN DER BCD-
C15E 29 0F    2EC3              AND #$0F            ;ZAHL,LOESCHEN DES
C160          2ED8 ;                MSB
C160          2EDB ;
C160 09 30    2F00 BCDASC1      ORA #$30            ;DAZUODERN VON $30
C162          2F28 ;                ERZEUGT (WEIL NUR ZAHL
C162          2F4F ;                ZWISCHEN 0 UND 9) DEN
C162          2F78 ;                ASCII-WERT ($30 BIS 39)
C162          2F7B ;
C162 91 62    2F98              STA (FAC1),Y        ;EINTRAGEN IN
C164          2FB7 ;                STRINGTABELLE
C164          2FBA ;
C164 C8       2FCC              INY                 ;ZAEHLER +1
C165          2FCF ;
C165 60       2FD5              RTS
C166          2FD8 ;
C166          3001 ;****** REST DES UP ASCII-BCD *********
C166          3004 ;
C166 20 13 C1 3020 TEST         JSR TEST1           ;PRUEFT AUF
C169          3042 ;                ASCII-ZAHL (0-9)
C169 60       3048              RTS
C16A          304B ;
C16A          3074 ;****** NMI-REAKTION AUF ALARM ********
C16A          3077 ;
C16A A9 77    3097 ALARM        LDA #L,ALIRQ        ;NEUER IRQ-
C16C 8D 14 03 30AB              STA IRQVL           ;VEKTOR
C16F A9 C1    30BA              LDA #H,ALIRQ
C171 8D 15 03 30C6              STA IRQVH
C174          30C9 ;
C174 4C BC FE 30E5              JMP NMIEND          ;REST DER NOR-
C177          3108 ;                MALEN NMI-ROUTINE
C177          310B ;
C177          3134 ;****** DIE  NEUE IRQ-ROUTINE *********
C177          3151 ;            RAHMENBLINKEN
C177          3154 ;
C177          3157 ;
C177 C6 02    3175 ALIRQ        DEC VERZ            ;ZEITSCHLEIFE
C179 F0 03    3191              BEQ BLINK           ;BLINKEN WENN 0
C17B          3194 ;
C17B 4C 31 EA 31B2              JMP NORM            ;SONST NORMALE IRQ
C17E          31B5 ;
C17E A5 04    31D5 BLINK        LDA VORW            ;ZAEHLER RUECK-
C180 85 02    31E8              STA VERZ            ;SETZEN
C182          31EB ;
C182 AD 20 D0 3203              LDA RAND            ;RAHMENFARBE
C185 45 03    321B              EOR FARB            ;INVERTIEREN
C187 8D 20 D0 3226              STA RAND
C18A          3229 ;
C18A 4C 31 EA 3244              JMP NORM            ;ZUM NORMAL-IRQ
C18D          3247 ;
C18D          324D              .EN
```

Listing 11. Der Quelltext zur Echtzeituhr (Schluß)



Effektives Programmieren in Assembler

Es gibt viele Möglichkeiten, ein Basic-Programm schneller und komfortabler zu gestalten. Aber auch für die Assemblerprogrammierung gibt es einige Tricks und Kniffe, die wir Ihnen in diesem praxisnahen Kurs verraten wollen.

Wer das Optimum an Geschwindigkeit aus seinem Computer herausholen will, kommt an Maschinensprache nicht vorbei. Die Grundlagen zur Maschinenprogrammierung wurden bereits im Kurs »Assembler ist keine Alchimie«, den Sie in diesem Sonderheft finden, geschaffen. Das Thema dieses Artikels ist es nun, die Möglichkeiten von Maschinensprache optimal zu nutzen. Sie erfahren, wie man

a) Programme beschleunigen und
b) Speicherplatz sparen kann.

Dazu werden Ihnen eine Vielzahl von Programmiertechniken, Tips und Tricks vermittelt, die Ihnen die Programmierung erleichern.

1. Beschleunigungen des Betriebssystems (in Assembler)

Der C 64 muß viele Aufgaben gleichzeitig erledigen: Bearbeiten des Hauptprogramms, Ablauf der Systeminterrupts und Senden des Video-Signals (an den Monitor/Fernseher). Alle diese Funktionen erfordern

- viele Zugriffe auf den Datenbus des Prozessors
- und dadurch Ausführungszeit.

Unser Grundproblem ist nun, wie wir den Computer dazu bewegen, diese Aufgaben nicht (oder nur teilweise) auszuführen.

a) Eingriffe in den Systeminterrupt

Eine detaillierte Beschreibung des Systeminterrupts finden Sie im bereits erwähnten Kurs »Assembler ist keine Alchimie«. Hier möchte ich nur zusammenfassen, was im normalen Interrupt des Betriebssystems geschieht: 60 mal in der Sekunde wird das Hauptprogramm verlassen und die Routine ab $EA31 angesprungen. Ist diese abgearbeitet, wird wieder ins Hauptprogramm zurückgesprungen. Während dieser Unterbrechung (»interrupt«) tut sich einiges:

- die RUN/STOP-Taste wird überprüft
- die Tastatur und der Datasettenmotor werden abgefragt
- das Cursorblinken wird erledigt
- die interne Uhr (TI$) wird gestellt.

Überlegen wir uns, welche Funktionen verzichtbar sind: Die RUN/STOP-Taste bewirkt nur in Basic-Programmen einen Abbruch, in Assembler müßte sie zum Beispiel über »JSR $FFE1« zusätzlich abgefragt werden. Die interne Uhr findet von Maschinensprache aus praktisch keine Verwendung. Kurz und gut, ein Maschinenprogramm kann auf beide Funktionen verzichten. Dies wird durch ein

```
LDA #$34
STA $0314
```

erreicht. Weil der Computer dadurch entlastet wird, läuft das Hauptprogramm etwas schneller ab.

Die Normaleinstellung erhält man mit

```
LDA #$31
STA $0314
```

> **Beschleunigungsmethode 1:**
> Trick: Verkürzung der Interrupt-Routine
> Nebenwirkungen: Abfrage der STOP-Taste und interne Uhr entfallen

Können Sie zwischenzeitlich auf die ganze Interrupt-Routine verzichten, genügt ein einziger Befehl:

```
SEI (»set interrupt«)
```

Er verhindert grundsätzlich das Auftreten von Interrupts. Die Normaleinstellung bewirkt:

```
CLI (»clear interrupt«)
```

> **Beschleunigungsmethode 2:**
> Trick: Interrupt total abschalten
> Nebenwirkungen: Abfrage von Tastatur, STOP-Taste und Datasette, sowie Cursor und interne Uhr entfallen.

Es gibt aber noch eine Möglichkeit, im Zusammenhang mit dem Systeminterrupt: Von der Adresse $DC05, die als Zähler dient, hängt die Anzahl der Interrupts (in der Regel 60 Aufrufe pro Sekunde) in einer bestimmten Zeit ab. Diese Adresse kann durch Schreibzugriff geändert werden. Schreibt man in $DC05 einen niedrigen Wert (im Extremfall 0), so werden sehr viele Interrupts ausgelöst. Dies macht sich in der Geschwindigkeit der Interrupt-Routine bemerkbar. Cursor und Tastaturabfrage werden sehr schnell, die interne Uhr geht vor, und so weiter. Verwendet man eine eigene, eventuell zeitkritische Interrupt-Routine, kann sie auf diese Weise beschleunigt werden.

Dieser Geschwindigkeitszuwachs geht allerdings auf Kosten des Hauptprogramms, das stark verlangsamt wird. Bei wenigen Interrupts (große Zahl in $DC05) wird es beschleunigt. Die entsprechenden Assemblerbefehle lauten:

```
LDA #$FF
STA $DC05
```

um eine starke Beschleunigung zu bewirken.

Die Normaleinstellung wird durch

```
LDA #$3A
STA $DC05
```

erreicht.

> **Beschleunigungsmethode 3:**
> Trick: Anzahl der Interruptaufrufe pro Sekunde ändern
> Nebenwirkungen: Bei zu wenigen Aufrufen hinken Uhr, Cursor und Tastaturabfrage nach; bei zu vielen werden sie zu schnell.

b) VIC-Register Nummer 17

Ist Ihnen schon bei Hypra-Load, beim Arbeiten mit der Datasette und einigen Kopierprogrammen aufgefallen, daß manchmal der Bildschirm abgeschaltet wird (ähnlich wie im FAST-Mode des C 128)? Dies kann man mit einem Vorhang vergleichen, der zwischenzeitlich den Bildschirm verdeckt. Der Bildschirm kann zwar nach wie vor (hinter dem Vorhang) geändert werden (PRINT-Anweisungen werden also ausgeführt), aber sichtbar wird die Wirkung erst, wenn der Vorhang entfernt wird.

Verantwortlich für das Ein-/Ausschalten des Bildschirms ist das VIC-Register Nummer 17:

Bit 4 gesetzt: Bildschirm wird angezeigt
Bit 4 gelöscht: Bildschirm wird abgeschaltet und nimmt Rahmenfarbe an.

Da wir die theoretischen Grundlagen haben, brauchen wir nur noch unser Wissen in Befehle umzusetzen:

Bildschirm abschalten:

```
LDA  $D011   ($D011 ist VIC-Register #17)
AND  #$EF    ($EF = %11101111)
STA  $D011                ↑
                        Bit 4
```

In diesem Zustand arbeiten manche Kopierprogramme um zirka 15% schneller. Programme, die nicht auf externe Geräte wie die Floppy zugreifen, laufen zirka 5% schneller ab.

Bildschirm wieder einschalten:

```
LDA  $D011
ORA  #$10    ($10 = %00010000)
STA  $D011                ↑
                        Bit 4
```

Dies ist der Normalzustand.

Beschleunigungsmethode 4:
Trick: Bildschirm abschalten
Nebenwirkungen: Der Bildschirminhalt ist nicht zu sehen, geht aber auch nicht verloren.

c) Hinweise zum bisher Gesagten

Alle bis zu dieser Stelle genannten Tricks beziehen sich auf die Beschleunigung von Programmen. Sie lassen sich leicht nachträglich einfügen, weil am Programmalgorithmus keine Änderungen erforderlich sind.

Sie können das Abschalten des Bildschirms mit dem Abschalten oder Einschränken des Interrupts verknüpfen, um die Geschwindigkeit noch weiter zu erhöhen. Wenn Sie den Interrupt ganz abschalten (SEI), bringt es keinen zusätzlichen Gewinn, ihn einzuschränken oder die Zahl der Aufrufe zu ändern.

Beachten Sie bitte, daß alle beschriebenen Tricks durch RUN/STOP-RESTORE, einem Reset oder den Assemblerbefehl BRK rückgängig gemacht werden.

2. Systembeschleunigungen in Basic

Hier erfahren Sie, wie sich die Systembeschleunigungen von Basic aus verwerten lassen. Die Nebenwirkungen bleiben allerdings diegleichen, wie unter 1. genannt.

a) Interrupt einschränken

POKE 788,52 verkürzt die Interrupt-Routine um das Abfragen der RUN/STOP-Taste und das Stellen von TI$.

POKE 788,49 Normalzustand

In Basic ist das Ausfallen von RUN/STOP und TI$ wesentlich störender als in Maschinensprache. Überprüfen Sie daher Ihre Programme auf Verwendung von TI$ und fügen Sie den POKE erst nach (!) der Fertigstellung des Programms ein.

b) Interrupt abschalten

POKE 56334,PEEK (56334) AND 254

schaltet den Interrupt ab,

POKE 56334,PEEK (56334) OR 1

schaltet ihn wieder ein. Dies geschieht dadurch, daß der Timer ab- beziehungsweise wieder eingeschaltet wird.

c) Anzahl der Interrupt-Aufrufe ändern

POKE 56325,0: Extrem viele Interruptaufrufe
POKE 56325,255: Extrem wenige (daraus folgt: Interrupt langsam, Basic-Programm schnell)

d) Bildschirm abschalten

POKE 53265,PEEK(53265) AND 239

schaltet den Bildschirm ab.

POKE 53265,PEEK(53265) OR 16

schaltet ihn wieder ein.

An dieser Stelle sei noch einmal auf Punkt 1c hingewiesen, damit keine (vermeidbaren) Probleme auftreten.

Anhand von Listing 1 wollen wir uns nun mit der Anwendung der Systembeschleunigungen befassen. Dieses kleine Beispielprogramm, an dem Sie nach Herzenslust experimentieren können, versucht, mit Hilfe von TI$ die Arbeitsdauer der Schleife (Zeile 150) zu messen.

Während des Ablaufs dieser Schleife, die kontinuierlich die Rahmenfarbe ändert, sollten Sie keine Taste drücken, um die Meßwerte nicht zu verfälschen.

Wenn Sie dies beachten, erhalten Sie folgende Werte:

1. Normalzustand: 000003
2. Verkürzter Interrupt: 000000
   An der gemessenen Zeit können Sie erkennen, daß TI$ abgeschaltet wurde.
3. Häufige Interrupts: 000010
   Aufgrund vieler Interrupt-Anforderungen wurde die Uhr TI$ sehr oft erhöht.
4. Seltene Interrupts: 000001
   Da die IRQ-Routine nur selten durchlaufen wurde, ist TI$ kaum weitergezählt worden.
5. Bildschirm abgeschaltet: 000002

Nur bei diesem Punkt (und natürlich auch bei »1«) hat TI$ volle Aussagekraft bezüglich der Ablaufzeit. An dieser Zeit können wir erkennen, daß durch das Abschalten des Bildschirms tatsächlich gegenüber »1« ein Zeitgewinn anfällt.

Bei den Punkten »3« und »4« wurde der Cursor eingeschaltet. Bei »3« (häufige Interrupts) ist er sehr schnell, bei »4« dagegen sehr langsam.

An Punkt »5« können Sie erkennen, daß bei abgeschaltetem Bildschirm der Hintergrund immer die Rahmenfarbe ($D020) annimmt, ohne daß wir die entsprechende Farbe ins Register $D021 »POKEn«.

```
90 GOTO 200                                    <026>
100 REM >> UP - SCHLEIFE <<                    <138>
110 :                                          <086>
120 PRINT" <TASTE>";:WAIT 198,1:POKE 198,0
    :FOR I=1 TO 7:PRINT CHR$(20);:NEXT         <221>
130 :                                          <106>
140 FOR I=1 TO 100:NEXT                        <122>
150 TI$="000000":FOR I = 0 TO 255:POKE 532
    80,I AND 15:NEXT:PRINT TI$:RETURN          <205>
160 :                                          <136>
170 REM >> UP - CURSORBLINKEN AUS <<           <206>
180 :                                          <156>
190 POKE 207,0:POKE 204,1:PRINT" ":RETURN      <186>
200 REM --------------------                   <222>
210 REM -- HAUPTPROGRAMM --                    <045>
220 REM --------------------                   <242>
230 :                                          <206>
240 PRINT CHR$(147)"DEMO FUER SYSTEMBESCHL
    EUNIGUNGEN (BASIC)";                       <061>
250 PRINT"-------------------------------
    ---------"                                 <127>
260 PRINT"{2DOWN}1) NORMALZUSTAND";:GOSUB
    100                                        <033>
270 :                                          <248>
280 PRINT"{DOWN}2) VERKUERZTER INTERRUPT";
    :POKE 788,52:GOSUB 100:POKE 788,49         <084>
290 :                                          <012>
300 PRINT"{DOWN}3) HAEUFIGE INTERRUPTS";:P
    OKE 56325,20:POKE 204,0:GOSUB 100:GOSU
    B 170                                      <018>
310 :                                          <032>
320 PRINT"4) SELTENE{2SPACE}INTERRUPTS";:P
    OKE 56325,150:POKE 204,0:GOSUB 100:GOS
    UB 170                                     <113>
330 SYS 64931:REM NORMALZUSTAND EIN            <253>
340 :                                          <062>
350 PRINT"5) BILDSCHIRM ABGESCHALTET ";:PO
    KE 53265,PEEK(53265) AND 239:GOSUB 140 <107>
360 POKE 53265,PEEK(53265) OR 16:PRINT"{DO
    WN}** ENDE **"                             <066>
```

© 64'er

Listing 1. Systembeschleunigungen in Basic

3. Optimierung der Bildschirmausgabe

Ohne die Bildschirmausgabe kommt kein Programm aus, aber oft kostet sie unnötig viel Rechenzeit. Der Grund ist hier nicht beim Betriebssystem zu suchen, sondern bei umständlicher Programmierung. Diese wiederum ist auf mangelndes Know-how zurückzuführen, welches wir nun ändern wollen.

In der Regel wird zur Ausgabe eines Zeichens dieses in den Akku geladen und die Routine BASOUT ($FFD2) aufgerufen. Veranschaulichen wir uns einmal die Arbeitsweise von BASOUT: Das Betriebssystem prüft bei jedem Zeichen, ob es sich um einen Buchstaben oder ein Steuerzeichen, zum Beispiel »Bildschirm löschen« handelt. Buchstaben werden in den Bildschirmcode umgewandelt und ins Bildschirm-RAM ab $0400 geschrieben.

Für Steuerzeichen existieren jeweils Unterroutinen die zum Beispiel eine Leerzeile einfügen, den Bildschirm löschen oder ähnliches.

Diese aufwendige Überprüfung verlangsamt die Bildschirmausgabe erheblich. BASOUT läßt sich zwar geringfügig beschleunigen, indem man statt bei $FFD2 (Kerneleinsprung) bei $E716 einsteigt, aber es geht noch schneller:

a) Bildschirm löschen

Langsam:

LDA #$93 $93 = 147 = Code für »Bildschirm löschen«, entspricht PRINT CHR$(147)
JSR $FFD2 (oder $E176)

Schnell:

JSR $E544 (Routine für »Bildschirm löschen«)

b) Cursor in Home-Position (linke obere Ecke)

Langsam:

LDA #$13 ; $13 = Code für »Cursor Home«
JSR $FFD2 (oder $E176)

Schnell:

JSR $E566 (Routine für »Cursor Home«)

c) Cursor-Positionierung

Langsam:

Senden von Steuerzeichen (CRSR DOWN, UP und so weiter) über BASOUT.

Schnell:

LDX #Zeile
LDY #Spalte
JSR $E50C (Cursorposition setzen)

d) Textausgabe

Unkomfortable Lösung:

Senden von Zeichen (Buchstaben, Grafikzeichen) über BASOUT.

Eine solche Schleife finden Sie in Listing 2, Zeilen 148 - 220 und 320 - 330. Nach dem Start durch »SYS 49152« gibt Listing 2 zweimal hintereinander den Text »DAS IST DER TEXT« aus. Das erste Mal wird der Text über eine BASOUT-Schleife gedrückt, beim zweiten Mal nimmt das Programm die Komfortable Lösung:

Ab der Adresse »TEXT« muß der Text (in ASCII-Darstellung) stehen, in dem keine Anführungszeichen vorkommen dürfen. Am Ende des Textes muß $00 als Endmarkierung zu finden sein. Die Ausgabe erfolgt dann über

LDA # < (TEXT) Low-Byte der Adresse
LDY # > (TEXT) High-Byte
JSR $AB1E

Die Routine $AB1E wird fortan als »STROUT« (STRing-OUTput = String-Ausgabe) bezeichnet. STROUT ist zwar etwas langsamer als BASOUT; dafür erlaubt die komfortable Parameterübergabe eine wesentlich bequemere Programmierung, wie Sie am zweiten Teil von Listing 2 (Zeilen 260 - 300, 320 - 330) sehen können. Mit nur drei Befehlen wird der Text ausgegeben!

Beschleunigungsmethode 5.

Zusammenfassung der bisherigen Alternativen zu BASOUT:

CLEAR HOME:	JSR $E544
CURSOR HOME:	JSR $E566
Cursorpositionierung:	LDX #Zeile LDY #Spalte JSR $E50C
Textausgabe:	Text ab TEXT ablegen (wie Listing 2, Zeile 320 - 330) LDA # < (TEXT) LDY # > (TEXT) JSR $AB1E

Alle diese Verfahren sind nicht nur schnell, sondern auch speicherplatzsparend.

Eine Anwendung von (fast) allen Routinen aus der Beschleunigungsmethode 5 zeigt Listing 3.

```
100  -.LI 1,3,0
110  -;
120  -; TEXTAUSGABE (UEBER BASOUT)
130  -;
140  -.BA $C000 ; START: SYS 49152
150  -;
160  -.GL BASOUT = $FFD2
170  -;
180  -          LDX #0
190  -SCHLEIFE LDA TEXT,X      ; ZEICHEN LESEN
200  -          INX
210  -          JSR BASOUT     ; UND AUSGEBEN
220  -          BNE SCHLEIFE   ; SCHON ENDMARKIERUNG?
230  -;
240  -; TEXTAUSGABE (UEBER STROUT)
250  -;
260  -.GL STROUT = $AB1E
270  -;
280  -          LDA #<(TEXT)   ; LOW-BYTE IN AKKU
290  -          LDY #>(TEXT)   ; HIGH-BYTE IN Y
300  -          JMP STROUT     ; TEXTAUSGABE UND ENDE
310  -;
320  -TEXT     .TX "DAS IST DER TEXT!"
330  -.BY 0 ; ENDMARKIERUNG DES TEXTES
```

Listing 2. Die unkomfortable Lösung, einen Text auszugeben

```
SEARCHING FOR $$

100  -.LI 1,3,0
110  -;
120  -; TEXTAUSGABE (UEBER STROUT)
130  -;
140  -.BA $C000 ; START: SYS 49152
150  -;
160  -.GL STROUT = $AB1E
170  -.GL CURSOR = $E50C
180  -.GL CLRSCR = $E544 ; BILDSCHIRM LOESCHEN
190  -;
200  -.GL ZEILE  = 12
210  -.GL SPALTE = 10
220  -;
230  -          JSR CLRSCR     ; = PRINT CHR$(147)
240  -          LDX #ZEILE     ; ZEILE IN X
250  -          LDY #SPALTE    ; SPALTE IN Y
260  -          JSR CURSOR     ; CURSOR SETZEN
270  -          LDA #<(TEXT)   ; LOW-BYTE IN AKKU
280  -          LDY #>(TEXT)   ; HIGH-BYTE IN Y
290  -          JMP STROUT     ; TEXTAUSGABE & ENDE
300  -;
310  -TEXT     .TX "DAS IST DER TEXT!"
320  -.BY 0 ; ENDMARKIERUNG FUER STROUT
```

Listing 3. Die komfortable Lösung einen Text auszugeben

Der Bildschirm wird gelöscht und in Zeile 12 ab Spalte 10 ein Text ausgegeben. Auch in diesem Programm sollten Sie zur Übung etwas experimentieren!

e) Kopieren des Textes in den Bildschirmspeicher

Dies ist die schnellste Methode: Der Text wird in den Bildschirmspeicher kopiert. Die lange Umwandlung entfällt völlig, da der Text als fertiger Bildschirmcode im Speicher abgelegt wird. Wenn einige Kopfzeilen (zum Beispiel mit Copyright-Vermerken) an verschiedenen Stellen ausgegeben werden sollen, ist es ratsam, ein kleines Unterprogramm zu erstellen. Dieses schreibt dann die Kopfzeilen direkt in den Bildschirmspeicher, ohne die aktuelle Cursor-Position zu beeinflussen.

Eines müssen Sie aber unbedingt beachten: Die Farbgebung ist nur durch Ändern des Farb-RAMs möglich.

Eine Tabelle der Bildschirmcodes finden Sie übrigens im Anhang des C 64-Handbuchs und am Schluß dieser Ausgabe.

Beschäftigen wir uns nun mit Listing 4:

Dieses Programm entspricht in der Wirkung Listing 3, gibt den Text jedoch nicht über die Betriebssystem-Routinen CURSOR und STROUT aus, sondern schreibt ihn direkt in den Bildschirm.

In den Zeilen 310 - 320 steht der Bildschirmcode des Textes.

Zurück zur Routine STROUT: Diese Routine arbeitet, da sie sich auf die BASOUT-Routine stützt, auch mit Peripheriegeräten wie Floppy und Drucker, wenn diese über dem CMD-Befehl als Ausgabegeräte definiert wurden. In »Assembler ist keine Alchimie« wurde gezeigt, wie man mit der BASOUT-Routine die Drucker-Ausgabe betreibt. Dort wurden alle wichtigen Routinen bis ins Detail beschrieben.

Listing 5 gibt einen Text zuerst auf dem Drucker und dann auf dem Bildschirm aus. Daran soll außer dem Druckerbetrieb auch gezeigt werden, wie man die Parameterübergabe an STROUT als Makro (Zeilen 230 - 270) definiert und sich somit einen bequemen Ausgabe-Befehl schafft.

```
100  -.LI 1,3,0
110  -;
120  -; TEXT IN VIDEO-RAM SCHREIBEN
130  -;
140  -.BA $C000 ; START: SYS 49152
150  -;
180  -.GL CLRSCR = $E544 ; BILDSCHIRM LOESCHEN
190  -;
200  -.GL ZEILE  = 12
210  -.GL SPALTE = 10
220  -;
230  -.GL VIDEORAM = 1024 ; BILDSCHIRMSPEICHER
240  -.GL ADRESSE = VIDEORAM + (40*ZEILE) + SPALTE
250  -;
255  -          JSR CLRSCR     ; = PRINT CHR$(147)
260  -          LDX #0
270  -SCHLEIFE  LDA TEXT,X     ; BILDSCHIRMCODE LESEN
280  -          BEQ ENDE       ; =0, DANN ENDE
290  -          STA ADRESSE,X  ; IN BILDSCHIRMSPEICHER
295  -          INX
296  -          JMP SCHLEIFE   ; NAECHSTES ZEICHEN
300  -ENDE      RTS
305  -;
310  -TEXT      .BY 4,1,19," ",9,19,20," "
311  -.BY 4,5,18," ",20,5,24,20,"!"
320  -.BY 0 ; ENDMARKIERUNG DES TEXTES
```

Listing 4. Die schnellste Lösung, einen Text auszugeben

```
100  -.LI 1,3,0
110  -;
120  -; DRUCKER-AUSGABE MIT
130  -; DER STROUT-ROUTINE
140  -;
150  -.GL STROUT = $AB1E
160  -.GL SETNAM = $FFBD ; DIE BEDEUTUNG
170  -.GL SETLFS = $FFBA ; DIESER ROUTINEN
180  -.GL OPEN   = $FFC0 ; ENTNEHMEN SIE
190  -.GL CHKOUT = $FFC9 ; BITTE DEM KURS
200  -.GL CLRCHN = $FFCC ; "ASSEMBLER IST
210  -.GL CLOSE  = $FFC3 ; KEINE ALCHIMIE"
220  -;
230  -.MA PRINT (ADRESSE)
240  -          LDA #<(ADRESSE)
250  -          LDY #>(ADRESSE)
260  -          JSR STROUT
270  -.RT
280  -;
290  -.BA $C000 ; START: SYS 49152
300  -;
310  -          LDA #0         ; KEINEN
320  -          JSR SETNAM     ; FILENAMEN
330  -;
340  -          LDA #4         ; LOG. FILENUMMER =4
350  -          TAX            ; GERAETEADRESSE 4
360  -          LDY #0         ; SEKUNDAERADRESSE 0
370  -          JSR SETLFS     ; PARAMETER SETZEN
380  -;
390  -          JSR OPEN       ; FILE OEFFNEN
400  -;
410  -          LDX #4         ; FILENUMMER 4
420  -          JSR CHKOUT     ; AUSGABE AUF DRUCKER LENKEN
430  -;
440  -...PRINT (TEXT) ; TEXT AUSGEBEN
450  -;
460  -          JSR CLRCHN     ; WIEDER BILDSCHIRMAUSGABE
470  -;
480  -...PRINT (TEXT) ; JETZT AUF BILDSCHIRM
490  -;
500  -          LDA #4         ; LOG. FILENUMMER 4
510  -          JMP CLOSE      ; FILE SCHLIESSEN
520  -; & PROGRAMM BEENDEN
530  -;
540  -TEXT      .TX "DIESER TEXT WIRD AUF"
550  -.TX " DEN DRUCKER AUSGEGEBEN !"
560  -.BY 13,13,13,0 ; 3 * CAR.RETURN
```

Listing 5. So gibt man Text auf dem Drucker aus

4. Unterprogramme

Ohne die Unterprogramm-Befehle JSR und RTS kommt fast kein Maschinenprogramm aus. Es ist allerdings ziemlich unbekannt, daß beide Befehle das Programm stark verlangsamen. Grund genug für uns, JSR und RTS näher zu betrachten:

Trifft der Prozessor auf JSR, schiebt er den aktuellen Programmzähler plus 2 (= Rücksprungadresse - 1) auf den Stack und springt dann zu der Adresse, die hinter JSR steht. Trifft er auf RTS, holt er die Adresse vom Stapel zurück, erhöht sie um 1 und verwendet sie wieder als Programmzähler.

Bemerkenswert ist, daß die Zugriffe auf den Stapel sich in keiner Weise von den Zugriffen über die Befehle PHA und PLA unterscheiden. Daher muß jedesmal der Stapelzeiger neu errechnet werden. Diese vielen Operationen sind schuld daran, daß JSR und RTS so langsam sind.

Da wir das Problem erkannt haben, können wir damit beginnen, unser Wissen anzuwenden.

a) Unterprogrammverschachtelung

Stellen wir uns folgendes Beispiel vor: ein Hauptprogramm ruft das Unterprogramm 1 auf. Dieses ruft an seinem Ende das Unterprogramm 2 auf, um dann mit RTS ins Hauptprogramm zurückzukehren.

Alles ziemlich schwierig, oder?

Deshalb gehen wir mit Hilfe einer Grafik vor: In Bild 1 sehen Sie ein Flußdiagramm nach obigem Aufbau. In der Beschriftung soll »Code« nicht »Kennwort« bedeuten, sondern heißt einfach »Befehlsnummer«.

Wie an den Pfeilen zu erkennen ist, werden zwei RTS-Befehle hintereinander abgearbeitet (von Unterprogramm 2 nach Unterprogramm 1 und von dort zum Hauptprogramm).

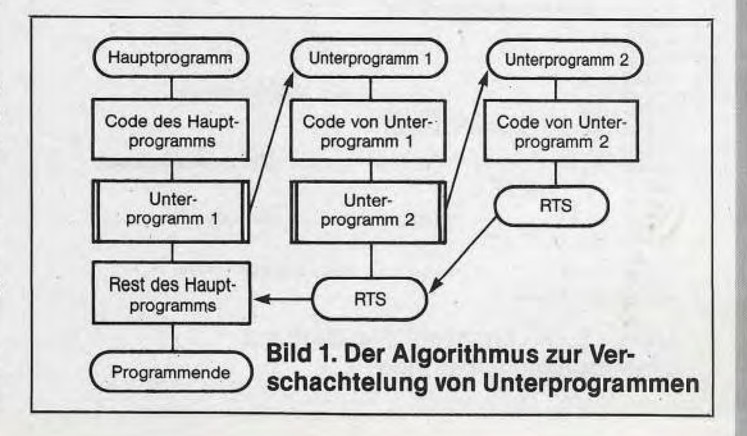


Bild 1. Der Algorithmus zur Verschachtelung von Unterprogrammen

Dies ist immer ein Indiz dafür, daß das Programm noch optimiert werden kann.

Eine »Übersetzung« von Bild 1 in Assembler ist Listing 6: Wenn Sie dieses über »SYS 49152« starten, ist aus den ausgegebenen Texten ersichtlich, welcher Programmteil wann abgearbeitet wird.

Sobald Sie die Struktur von Bild 1 beziehungsweise Listing 6 verstanden haben, können wir uns mit der optimierten Form befassen, die in Bild 2 beziehungsweise Listing 7 zu finden ist.

Hier wird das ehemalige Unterprogramm 2 ans Ende von Unterprogramm 1 gehängt (wobei es ebenfalls über JMP UP2 angesprungen werden könnte). Auf diese Weise muß es nicht über JSR aufgerufen werden, was auch einen RTS-Befehl überflüssig macht.

Trotz dieser Änderung kann das Unterprogramm 2 auch weiterhin als Unterprogramm aufgerufen werden, da bei JSR UP2 die CPU auf einen RTS-Befehl trifft (Bild 2).

In Listing 7 muß noch der JMP-Befehl in Zeile 480 erläutert werden:

Dort muß nicht JSR STROUT:RTS stehen, weil am Ende der STROUT-Routine im ROM ohnehin ein RTS steht. Deshalb benötigt unser Programm keinen eigenen RTS-Befehl zur Rückkehr ins Hauptprogramm.

Die folgende Regel gilt für Aufrufe von Betriebssystemroutinen:

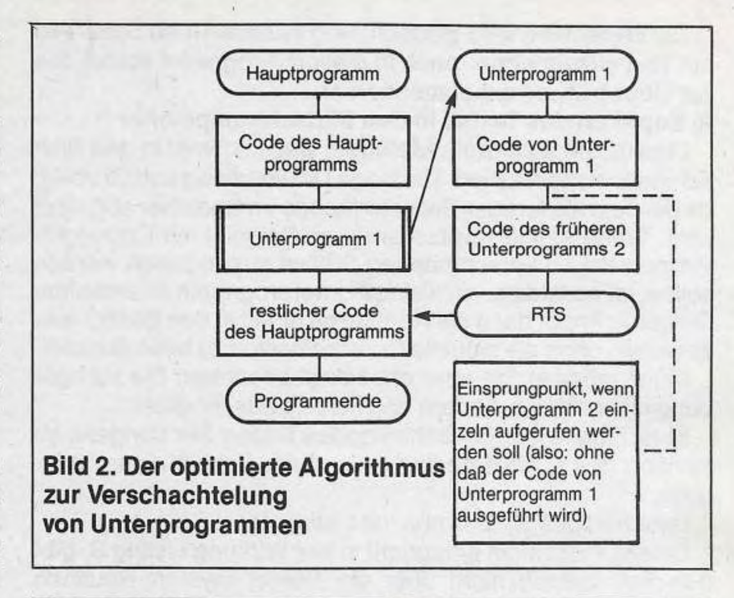


Bild 2. Der optimierte Algorithmus zur Verschachtelung von Unterprogrammen

JSR $XXXX **RTS**	**entspricht**	**JMP $XXXX**

Voraussetzung ist, daß im Unterprogramm ab $XXXX keine Stapelmanipulation erfolgt, wie sie gleich beschrieben wird. Das geschilderte Verfahren zur Unterprogrammverschachte-

```
100   -.LI 1,3,0
110   -.BA $C000 ; START: SYS 49152
120   -;
130   -; UNTERPROGRAMMVERSCHACHTELUNG
140   -; (OPTIMIERTE ASSEMBLERVERSION)
150   -;
160   -.GL STROUT = $AB1E
170   -;
180   -.MA PRINT (ADRESSE)
190   -          LDA #<(ADRESSE)
200   -          LDY #>(ADRESSE)
210   -          JSR STROUT
220   -.RT
230   -;
240   -; --------------- HAUPTPROGRAMM
250   -;
260   -...PRINT (TEXT1)
270   -;
280   -          JSR UP1
290   -;         ↑ AUFRUF VON UNTERPROGRAMM 1
300   -;
310   -...PRINT (TEXT2)
320   -;
330   -          JMP $A474       ; WARMSTART
340   -;
350   -;
360   -; --------------- UNTERPROGRAMM 1
370   -;
380   -UP1       NOP             ; BELIEBIGER CODE
390   -...PRINT (TEXT3)
400   -;
410   -;
420   -;
430   -; --------------- CODE VON UNTERPROGRAMM 2
440   -;
450   -UP2       NOP             ; BELIEBIGER CODE
460   -          LDA #<(TEXT4)   ; LOW-BYTE
470   -          LDY #>(TEXT4)   ; HIGH-BYTE
480   -          JMP STROUT      ; TEXTAUSGABE
490   -; UND RUECKSPRUNG VOM UNTERPROGRAMM,
500   -; WEIL AM ENDE DER STROUT-ROUTINE
510   -; EIN RTS-BEFEHL STEHT.
10000-;
10010-;
10020-; --------------- TEXTE
10030-;
10040-TEXT1     .TX "HIER IST DAS HAUPTPROGRAMM."
10050-.BY 13,13 ; 1 LEERZEILE
10060-.BY 0 ; ENDMARKIERUNG
10070-;
10080-TEXT2     .TX "HIER IST WIEDER DAS HAUPTPROGRAMM."
10090-.BY 13,13,0
10100-;
10110-TEXT3     .TX "HIER IST DAS UNTERPROGRAMM 1."
10120-.BY 13,13,0
10130-;
10140-TEXT4     .TX "HIER IST DAS UNTERPROGRAMM 2."
10150-.BY 13,13,0
```

Listing 6. Die umständliche Methode, Unterroutinen aufzurufen

```
100   -.LI 1,3,0
110   -.BA $C000 ; START: SYS 49152
120   -;
130   -; UNTERPROGRAMMVERSCHACHTELUNG IN ASSEMBLER
140   -;
150   -.GL STROUT = $AB1E
160   -;
170   -.MA PRINT (ADRESSE)
180   -          LDA #<(ADRESSE)
190   -          LDY #>(ADRESSE)
200   -          JSR STROUT
210   -.RT
220   -;
230   -; --------------- HAUPTPROGRAMM
240   -;
250   -...PRINT (TEXT1)
260   -;
270   -          JSR UP1
280   -;         ↑ AUFRUF VON UNTERPROGRAMM 1
290   -;
300   -...PRINT (TEXT2)
310   -;
320   -          JMP $A474       ; WARMSTART
330   -;
340   -;
350   -; --------------- UNTERPROGRAMM 1
360   -;
365   -UP1       NOP             ; BELIEBIGER CODE
370   -...PRINT (TEXT3)
380   -;
390   -          JSR UP2
400   -;         ↑ AUFRUF VON UNTERPROGRAMM 2
410   -;
420   -          RTS             ; UP1 VERLASSEN
430   -;
440   -;
450   -; --------------- UNTERPROGRAMM 2
460   -;
465   -UP2       NOP             ; BELIEBIGER CODE
470   -...PRINT (TEXT4)
480   -;
490   -          RTS             ; UP2 VERLASSEN
500   -;
10000-;
10010-; --------------- TEXTE
10020-;
10030-TEXT1     .TX "HIER IST DAS HAUPTPROGRAMM."
10040-.BY 13,13 ; 1 LEERZEILE
10050-.BY 0 ; ENDMARKIERUNG
10060-;
10070-TEXT2     .TX "HIER IST WIEDER DAS HAUPTPROGRAMM."
10080-.BY 13,13,0
10090-;
10100-TEXT3     .TX "HIER IST DAS UNTERPROGRAMM 1."
10110-.BY 13,13,0
10120-;
10130-TEXT4     .TX "HIER IST DAS UNTERPROGRAMM 2."
10140-.BY 13,13,0
```

Listing 7. Die optimierte Methode, Unterroutinen aufzurufen

lung und die entsprechenden Regeln können Sie dann auf jede (!) Programmiersprache übertragen.

b) Stapelmanipulation

Wenn Sie »Exbasic Level II« kennen, wissen Sie sicher den Befehl »DISPOSE RETURN« zu schätzen. Er dient dazu, ein Unterprogramm ohne RETURN abzuschließen. Dadurch kann dieses zum Beispiel über GOTO verlassen werden.

In Assembler ist dies auch möglich. Die Befehlseingabe

```
PLA
PLA
```

entspricht in der Wirkung »DISPOSE RETURN«.

Da die Rücksprungadresse auf den Stapel abgelegt wird und dort 2 Byte in Anspruch nimmt, kann sie über PLA:PLA wieder vom Stapel geholt werden. Ein Unterprogramm ist nach PLA:PLA eigentlich kein Unterprogramm mehr, sondern Bestandteil des aufrufenden Programms. PLA:PLA findet vor allem in der Fehlerbehandlung Anwendung. An einem späteren Listing werden wir dies noch sehen. Nach PLA:PLA kann ein Unterprogramm über JMP verlassen werden. Dies machen wir uns zunutze, um den Rücksprung an eine beliebige Adresse zu simulieren. Dies ist sonst nicht möglich, da bei RTS immer hinter den Befehl gesprungen wird, der das Unterprogramm aufgerufen hat.

Ein RTS an eine beliebige Adresse müßte »RTS XXXX« heißen, doch diesen Befehl gibt es beim 6510 nicht. So wird er aber simuliert:

```
PLA          ; holt Rücksprungadresse
PLA          ; vom Stapel und
JMP $XXXX    ; springt nach $XXXX
```

So sieht ein Makro dazu aus:

```
-.MA    RTS (RUECKSPRUNGADRESSE)
-       PLA
-       PLA
-       JMP RUECKSPRUNGADRESSE
-.RT
```

Und noch ein Mangel der Unterprogrammbefehle soll beseitigt werden: Obwohl es JMP (indirekt) gibt, kennt der 6510 keinen Befehl wie JSR (indirekt); über Stapelmanipulation ist dies dennoch möglich (siehe dazu auch im 64'er, Ausgabe 1/86: Assembler-Bedienung leicht gemacht).

Nehmen wir an, im Vektor $14/$15 steht die Adresse $C000. Nun soll über den $14/$15-Vektor ein Unterprogramm aufgerufen werden (also das ab $C000). Bild 3 zeigt, was im einzelnen geschehen muß.

Die Rücksprungadresse steht zwar in Bild 3 direkt hinter dem JMP ($0014)-Befehl, kann aber auch anderswo im Programm stehen.

Folgendes Makro ermöglicht die Simulation von JSR (indirekt):

```
-. MA JSRIND (VEKTOR, RUECKSPRUNGADRESSE)
-       LDA # >(RUECKSPRUNGADRESSE-1)
-       PHA
-       LDA # <(RUECKSPRUNGADRESSE-1)
-       PHA
-       JMP (VEKTOR)
-. RT
```

Diese Simulation von JSR ($XXXX) verwendet auch der SYS-Befehl (disassemblieren Sie von $E12A bis $E155 und betrachten Sie dazu Bild 3).

Zuerst holt er die Zahl nach SYS in die Adressen $14/$15, dann legt er die Rücksprungadresse ($E147) -1 auf dem Stack ab. Nun holt er die Register P, A, X, Y aus den Adressen $030F, $030C, $030D, $030E. Es folgt ein indirekter Sprung über $0014/$0015.

Nach dem Rücksprung werden die Register wieder im Speicher dort abgelegt, woher sie genommen wurden und ein Sprung ins Basic wird durchgeführt.

Später werden wir noch eine weitere Möglichkeit für JSR (ind) kennenlernen, die aber nicht auf Stapelmanipulation beruht.

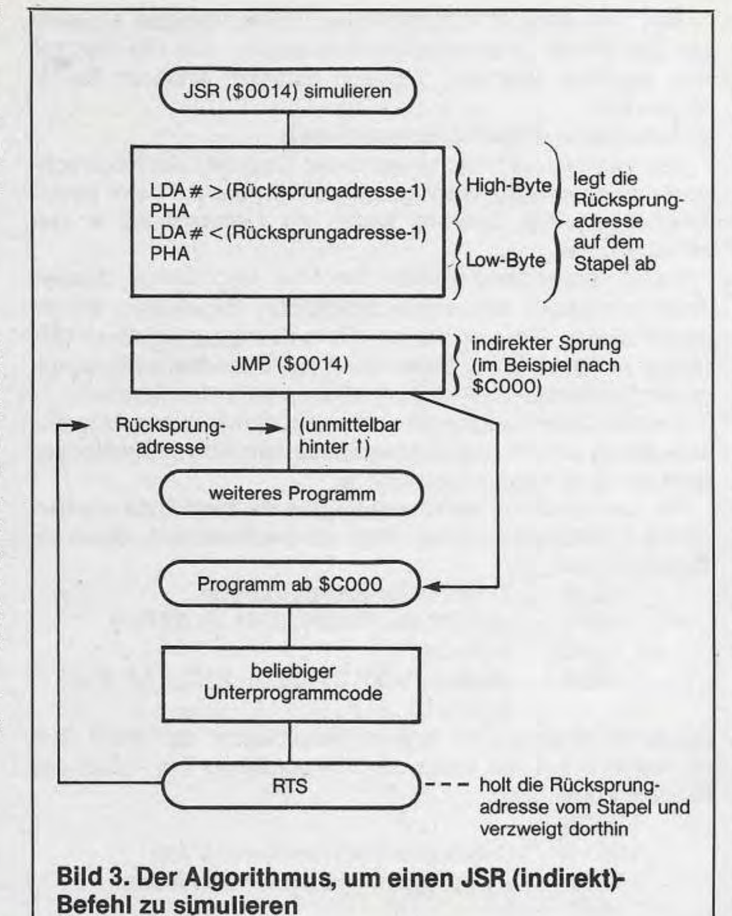


Bild 3. Der Algorithmus, um einen JSR (indirekt)-Befehl zu simulieren

c) Vergleich zwischen Unterprogramm und Makro bezüglich Geschwindigkeit

Wenn Sie den Hypra-Ass (oder einen anderen Makro-Assembler) besitzen, haben Sie die Möglichkeit, Befehlsfolgen als Makros zu definieren. Makros sind deswegen so beliebt, weil sie den größten Vorteil von Unterprogrammen bieten, nämlich Übersichtlichkeit. Da Makros aber wie »normale« Befehle im Speicher stehen, entfällt der Aufruf über JSR und RTS. Dies ist der Grund, weshalb Makros etwas schneller (wenige Taktzyklen) als Unterprogramme sind. Das Problem, wann Makros und wann Unterprogramme vorteilhaft sind, wird später noch aufgegriffen.

5. Tabellen

Im allgemeinen Sprachgebrauch werden Tabellen als »geordnete Zusammenstellungen von Daten« verstanden. Diese Funktion haben sie auch in Computerprogrammen, wo man sie daran erkennt, daß Tabellen keinen Befehlscharakter haben.

SMON-Benutzer können mit »FT« ein Programm nach Tabellen durchsuchen lassen; dann sucht SMON im Programm nach Bytes, die nicht zu Maschinensprachebefehlen gehören.

Wozu werden nun Tabellen verwendet?

In der Regel dienen Tabellen einem Computerprogramm als »elektronischer Rechenschieber«. So wie das Kopfrechnen durch einen Rechenschieber ersetzt werden kann, weil man nur in einer geordneten Zusammenstellung von Ergebnissen das richtige suchen muß, kann ein Programm aus seinen Tabellen denselben Nutzen ziehen: die Berechnungen entfallen, die Programmierung wird einfacher.

Aus den weniger erforderlichen Berechnungen entsteht ein deutlicher Geschwindigkeitszuwachs, der Hauptvorteil von Tabellen. Wie man Tabellen einsetzt, erfahren Sie im folgenden.

a) Tabellen aus Rechenergebnissen

Noch einmal zum Rechenschieber. Es geht beim Kopfrechnen viel schneller, 4x10 auszurechnen als 4x7. Bei einem Rechenschieber besteht kaum ein Unterschied in der »Rechenzeit«.

Dementsprechend existiert fast kein Algorithmus, dessen Ausführungszeit bei unterschiedlichen Parametern immer gleich bliebe. Wer den Artikel »Dem Klang auf der Spur (5)« (64'er, Ausgabe 5/85, Seite 152 ff.) gelesen hat, weiß, welch grobe Differenzen bei Multiplikationen auftreten können.

Ersetzt (beziehungsweise unterstützt) man einen Algorithmus durch eine Multiplikationstabelle, fällt eine einheitlichere (und kürzere) Ausführungszeit an.

Für das Rechnen mit einzelnen Bits in einem Byte werden oft die Zweierpotenzen benötigt; es empfiehlt sich, diese als Tabelle anzulegen:

```
1000 -; Zweierpotenzen als Tabellle
1010 -; im DOS der Floppy 1541 ab $EFE9
1020 -; zu finden
1030 -; ZWEIPOT .BY 2↑0, 2↑1, 2↑2, 2↑3, 2↑4,
        2↑5, 2↑6, 2↑7
```

Folgende Unterroutine legt im Akkumulator den Wert 2↑A ab, wobei mit A der Inhalt des Akkumulators bei Aufruf der Routine gemeint ist:

```
10000 -;
10010 -; Subroutine zur Berechnung von
10020 -; 2↑A (Ergebnis kommt in den Akku)
10030 -;
10040 - TAX ; Akku in Indexregister
10050 - LDA ZWEIPOT,X ; aus Tabelle einlesen
10060 - RTS ; Das war's schon! Wer ein
        schnelleres und zugleich so einfaches
        Verfahren kennt, möge sich melden...
10070 - ZWEIPOT
        .BY 2↑0,2↑1,2↑2,2↑3,2↑4,2↑5,2↑6,2↑7
```

Wenn A größer als 7 ist, liefert das Programm falsche Werte. Sie können es noch erweitern, wenn Sie es für nötig halten.

```
100  -.LI 1,3,0
110  -.BA $C000 ; START: SYS 49152
120  -;
130  -; RECHNUNG MIT FLIESSKOMMAWERTEN
140  -;
150  -.GL MEMFAC = $BBA2
160  -.GL FACOUT = $AABC
170  -.GL SQRFAC = $BF71
180  -.GL LOGNAT = $B9EA
190  -;
200  -.MA HOLE (ADRESSE) ; MAKRO-DEF.
210  -           LDA #<(ADRESSE); HOLT MFLPT-ZAHL
220  -           LDY #>(ADRESSE); VON ADRESSE IN
230  -           JSR MEMFAC      ; DEN FAC
240  -.RT
250  -;
260  -;
270  -...HOLE (BSPZAHL)
280  -;
290  -           JSR FACOUT      ; AUSDRUCKEN
300  -;
310  -...HOLE (BSPZAHL)
320  -;
330  -           JSR SQRFAC      ; QUADRATWURZEL
340  -;
350  -           JSR FACOUT      ; AUSDRUCKEN
360  -;
370  -...HOLE (BSPZAHL)
380  -;
390  -           JSR LOGNAT      ; LOGARITHMUS NATURALIS
400  -;
410  -           JMP FACOUT      ; AUSDRUCKEN
500  -;
510  -; BEISPIELZAHL 1.23456
520  -; IM MFLPT-FORMAT
530  -;
540  -BSPZAHL   .BY $81,$1E,$06,$0F,$E5
550  -;
```

Listing 8. Fließkommazahlen in Assembler verarbeiten

b) Tabellen aus Fließkommawerten

Zu den zeitraubendsten Operationen gehört die Rechnung mit Fließkommazahlen. Daß diese selbst in Maschinenprogrammen lähmend wirkt, sehen Sie am HiRes-3-Befehl »FUNKT« (64'er, Ausgabe 3/85, Grafikkurs-Anwendung). Daher sollte man nur dann auf die Fließkommaroutinen zugreifen, wenn es unvermeidbar ist. Berechnen Sie soviele Werte wie möglich voraus, hierfür eignet sich der Direktmodus des Basic-Interpreters besonders gut! Wie Sie einen auf diese Weise berechneten Wert ins MFLPT-(Floating Point)Format umwandeln können, zeigt Ihnen der folgende Kasten.

Verfahren zur Umwandlung einer Zahl ins MFLPT-Format

1. SMON (oder anderen Monitor) laden
2. RESET auslösen oder NEW eingeben
3. »XX = Fließkommazahl« eingeben, zum Beispiel »XX = 1.23456«
4. Monitor starten (SYS 49152)
5. »M 0805 0809« eingeben

Sie sehen nun in den Adressen $0805 - $0809 die MFLPT-Darstellung der Zahl, mit der Sie die Variable XX belegt haben.

Damit wir uns unter Zuhilfenahme präziser Fachausdrücke und Abkürzungen verständigen können, sollten Sie den Abschnitt in »Assembler ist keine Alchimie« aufmerksam lesen, der sich mit Fließkommazahlen befaßt. Nach dem Studium dieses Abschnitts sollten Ihnen Begriffe wie »MFLPT«, »FAC« oder »ARG« geläufig sein.

Im Falle der Zahl 1.23456 erhalten wir als Ergebnis:

:0805 81 1E 06 0F E5...

Diese Werte legen wir folgendermaßen als Tabelle ab:

540 -BSPZAHL .BY $81, $1E, $06, $0F, $E5

Wie wir nun diese Zahl verarbeiten, zeigt Ihnen Listing 8. Das Makro (200 - 240) stützt sich auf die Interpreter-Routine MEMFAC, die eine Zahl (Adresse wird in Akku/Y-Register übergeben) vom Speicherformat MFLPT in den FAC als FLPT-Zahl schreibt und dabei die erforderliche MFLPT-→FLPT-Umwandlung durchführt.

In der Tabelle in Zeile 540 können Sie beliebige Fließkommawerte (sofern Sie diese wie angegeben berechnet haben) einsetzen, das Programm rechnet dann mit der jeweiligen Fließkommazahl, die ab BSPZAHL im MFLPT-Format steht.

Diese Zahl wird zunächst nur in den FAC geladen und der FAC wird dann ausgedruckt (270 - 290), dann wird die Zahl wieder geholt, die Wurzel berechnet und ausgegeben (310 - 350). Schließlich wird die Zahl wieder in den FAC geholt, der natürliche Logarithmus errechnet und auch ausgegeben (370 - 410).

Zur Routine FACOUT sind, außer daß sie den Inhalt des FAC ausgibt, noch zwei Bemerkungen zu machen:

1. Nach der Zahl wird noch ein CARRIAGE RETURN ausgegeben.
2. Nach dem Aufruf von FACOUT hat sich der Inhalt des FAC aufgrund mehrerer Divisionen durch Zehnerpotenzen verändert.

Auf das Thema »Fließkommaarithmetik« geht Texteinschub 1 noch näher ein. Dort werden auch weitere Interpreter-Routinen vorgestellt.

c) Sprungtabelle

Beim Thema »Unterprogramme« wurde Ihnen eine Methode vorgestellt, um JSR (ind) zu simulieren. Diese erweist sich in Verbindung mit einer Tabelle, in der die Sprungadressen gespeichert sind, als sehr nützlich. So kann beispielsweise eine Parallele zum Basic-Befehl ON...GOSUB ZIEL1,ZIEL2.... geschaffen werden.

Ein Beispiel: Wenn der Basic-Interpreter auf einen Basic-Befehl trifft, holt er aus der Tabelle $A00C - $A09D die Adresse der zugehörigen Routine. Diese springt er dann durch Stapelmanipulation an.

Der SMON arbeitet genauso: Seine Sprungtabelle liegt im Bereich $C02B - $C06B.

Die Anwendung von Sprungtabellen werden wir noch ausführlich im folgenden Abschnitt d) sowie bei der Besprechung von Listing 11 behandeln.

d) Vergleichstabellen

Weder der SMON noch der Basic-Interpreter benutzen zum Suchen der zum jeweiligen Befehl gehörenden Routine eine Reihe von CMP-Abfragen mit BRANCH-Befehlen. Auch für die Vergleichswerte (in diesem Fall die Befehlswörter) gibt es eine Tabelle: Beim SMON liegt sie im Bereich $C00B - $C02A, beim Basic-Interpreter $A09E - $A327.

Sprung- und Vergleichstabellen sind in gleicher Befehlsfolge angeordnet; wird der Befehl an einer bestimmten Stelle in der Vergleichstabelle gefunden, erfolgt ein Sprung an die Adresse, die an gleicher Stelle in der Sprungtabelle steht.
So sehen die Befehls- und Vergleichstabellen im SMON aus:

Spalte Nr.	1	2	3	4	....
Befehl	/	#	$	%	....
Sprungadr. $	CADB	C920	C908	C91C	....

Die Sprungadressen sind wegen der Stapelmanipulation in der Tabelle ab $C02B um 1 dekrementiert gespeichert; in der Darstellung sehen Sie aber das tatsächliche Sprungziel.

Wir werden jetzt anhand des SMON die Verwendung einer Vergleichs-Sprungtabelle in Assembler erläutern.

Wenn wir die zum Befehl »#« gehörende Sprungadresse finden wollen, gehen wir folgendermaßen vor:

1. Wir suchen in Reihe 2 das #-Zeichen.
2. Wir gehen (in derselben Spalte) eine Reihe nach unten und finden dort die Sprungadresse ($C92C).

Der Computer hat nicht die Möglichkeit, direkt eine Reihe weiter unten die Suche fortzusetzen. Er muß einen Umweg wählen und sich die Spalte merken. Ein Beispiel:

1. Der SMON sucht unter den Elementen aus Reihe 2 das »#«. In einem Zähler merkt er sich die Spalte, in der der Befehl gefunden wurde.
2. Nun sucht er in Reihe 3 in der Spalte, die im Zähler steht, die zugehörige Sprungadresse.

Wie ähnlich beide Suchvorgänge sind, erkennen Sie daran, daß jedesmal die Hauptschritte 1. und 2. vorkommen.

Nach so viel Theorie sehen wir uns nun umso ausführlicher die Routine im SMON an, die für die Steuerung der Vergleichs-Befehlstabelle verantwortlich ist. Dazu können Sie »D C303 C323« eingeben.

Bei Adresse $C303 steht im Akku der ASCII-Code des Kommandos, das der SMON ausführen soll (zum Beispiel $40, wenn ein M-Befehl eingegeben wurde).

C303 LDX #$20 — 32–1 Befehle müssen durchsucht werden. Weshalb »–1« erforderlich ist, liegt an der Schleifenstruktur und ist unbedeutend.

C305 CMP $C00A,X — Akku (enthält Befehl) mit X-tem Element der Befehlstabelle vergleichen; $C00A = Befehlstabelle –1, weil Adresse $C00A nie zum Vergleich herangezogen wird.

C308 BEQ $C30F — Vergleich positiv; im X-Register steht jetzt die Spalte.

C30A DEX — Zähler wird dekrementiert; es handelt sich hier um eine »Dekrementierschleife« (dieses Thema wird noch behandelt).

C308 BNE $C305 — Wenn der Zähler noch nicht gleich 0 ist, folgt ein Sprung zum Schleifenbeginn.

C30D BEQ $C2D1 — Wenn X=0, dann wurde die ganze Tabelle durchsucht, und der Befehl nicht gefunden! Deshalb wird in die SMON-Fehlerbehandlung gesprungen.

C30F JSR $C315 — Diese Stelle wird von $C308 aus angesprungen; hier wiederum steht ein Aufruf des Unterprogramms ab $C315, das etwas weiter unten besprochen wird.

C312 JMP $C2D6 — Nachdem nun der Befehl durch die Subroutine $C315 abgearbeitet wurde, folgt ein Sprung zur Eingabe des nächsten Befehls.

C315 TXA — Das ist sie, die Subroutine! Weil im X-Register die Nummer des Befehls (= Spalte in Tabelle) steht, kommt das X-Register ins Hauptrechenregister.

C316 ASL — Die Befehlsnummer wird mit 2 multipliziert...

C317 TAX — und kommt wieder ins X-Register. Die Multiplikation mit 2 ist erforderlich, weil in der Sprungtabelle ein Element doppelt so lang ist, wie in der Vergleichstabelle, nämlich 2 Byte. Die Sprungadressen belegen deshalb 2 Byte, weil sie aus Low- und High-Bytes bestehen.

C318 INX — Das X-Register wird um 1 erhöht, da das High-Byte eine Position hinter dem Low-Byte steht.

C319 LDA $C029, X — High-Byte wird gelesen. Die Sprungtabelle beginnt zwar 2 Byte nach $C029, aber weil es keine Spalte 0 gibt, muß der Speicherbedarf einer Sprungadresse (=2) abgezogen werden.

C31C PHA — Das High-Byte der Adresse wird auf den Stapel gelegt.

C31D DEX — -1, weil Low-Byte eine Adresse vor High-Byte steht.

C31E LDA $C029,X — Nun wird auch das Low-Byte der Adresse

C321 PHA — auf den Stapel geschoben.

C322 RTS — Der Befehl RTS wird hier zur Simulation von JMP (ind) verwendet. Auf dieses (unpraktische) Verfahren soll nicht weiter eingegangen werden, weil der 6510 den Befehl JMP (ind) kennt. Wichtig ist für uns nur, daß jede SMON-Routine mit einem RTS abgeschlossen wird, dann erfolgt ein Rücksprung zur Adresse $C312.

Damit haben wir SMONs Schleife zum Suchen eines Befehls und dessen Routine durchleuchtet. Sofern Sie ein ROM-Listing zur Verfügung haben, können Sie zusätzlich die entsprechenden Stellen im Basic-Interpreter ansehen. Dieser aber benötigt wegen seiner unterschiedlich langen Befehle einen etwas komplizierteren Suchalgorithmus, was wiederum zu erheblich höherer Ausführungszeit beiträgt.

6. Vergleiche von Prüfsummen

Nun lernen wir ein besonders raffiniertes Vergleichsverfahren kennen:

Wie gesagt, benötigen Vergleiche mit Wörtern, die aus unterschiedlich vielen Zeichen bestehen, mehr Taktzyklen. Dies wäre nicht so, wenn wir alle Zeichen auf eine einheitliche Länge bringen würden. Genau dies tut der Basic-Interpreter: Bei Eingabe einer Zeile wandelt er alle Basic-Befehlswörter in Token um. Jedes Token vertritt einen Befehl und kann, da es nur ein Byte benötigt, schneller erkannt werden, als es bei mehreren Bytes möglich wäre.

Ein Nachteil ist jedoch der Speicherplatzaufwand; für die Umwandlung müssen die Befehle irgendwo im Speicher in Langform vorhanden sein.

Es gibt aber noch ein anderes Verfahren, einer Zeichenkette einen Wert zuzuweisen: Die Prüfsummenberechnung. Diese führen zum Beispiel die Eingabehilfen »Checksummer« und »MSE« durch: Aus 8 Byte Programmcode und 2 Byte Adresse errechnet der MSE eine 1 Byte Prüfsumme.

In Bild 4 sehen Sie einen sehr zuverlässigen Algorithmus zur Berechnung von Prüfsummen (insofern zuverlässig, als er sehr unterschiedliche Prüfsummen ermittelt). Listing 9 stellt ein Hilfsprogramm dar, das zu einer Eingabe die Prüfsumme nach dem Algorithmus aus Bild 4 errechnet.

In Listing 9 ist Ihnen eventuell die Routine NUMOUT nicht bekannt. Daher eine Kurzbeschreibung: NUMOUT gibt eine positive Integerzahl, die im Akkumulator (High-Byte) und im X-Register (Low-Byte) übergeben wird, aus. NUMOUT wird zum Beispiel von der LIST-Routine bei der Ausgabe einer Zeilennummer aufgerufen.

Die Routine BASIN soll ebenfalls erklärt werden, da sie in allen folgenden Programmen verwendet werden wird. Wenn die Routine BASIN zum ersten Mal aufgerufen wird, erwartet das Betriebssystem eine Eingabe (normalerweise von Tastatur), die der Eingabe einer Basic-Zeile entspricht. Nach der Eingabe wird das erste eingegebene Byte in den Akku geladen, jeder weitere Aufruf von BASIN holt das nächste Zeichen in den Akku. Wurden alle Bytes eingelesen, wird im Akku der Wert 13 ($0D, RETURN) übergeben. Danach führt ein weiterer Aufruf von BASIN zu erneuter Eingabe von Tastatur.

Ein großer Vorteil von Prüfsummen ist, daß die Vergleiche mit nur einem Byte, nämlich der Prüfsumme, durchgeführt werden müssen.

Wie man in den Genuß dieses Vorteils kommt, zeigt Listing 10. Wenn Sie den Namen eines Computers (C 64, VC 20, PC 128 oder AMIGA) eingeben, nennt das Programm den in diesem Computer installierten Mikroprozessor. Bei der Eingabe der Computernamen kann man aufgrund der Zeilen 230 und 248 beliebig viele Leerzeichen eingeben. Bei der Errechnung der Prüfsummen mit Listing 9 dürfen allerdings keine eingegeben werden, da Listing 9 diese nicht überliest und somit ein falsches Ergebnis liefern würde.

Der Programmteil, der die Prüfsumme der Eingabe berechnet, ist mit Ausnahmen der Zeilen 230/240 aus Listing 9 übernommen worden. Nach Zeile 450 wird die ermittelte Prüfsumme mit der Tabelle »PRÜFSUMMEN« (Zeile 2060) verglichen.

Bei »WEITER2« (Zeile 620) steht im X-Register die Spalte,

```
100  -.LI 1,3,0
110  -.BA $C000 ; START: SYS 49152
120  -;
130  -.GL BASIN = $FFCF
140  -.GL NUMOUT = $BDCD
150  -.GL STROUT = $AB1E
160  -;
170  -ANFANG    LDA #<(TEXT1)
180  -          LDY #>(TEXT1)
190  -          JSR STROUT
200  -;
210  -          LDX #0
220  -SCHLEIFE1 JSR BASIN
230  -          CMP #13         ; 13 = RETURN
240  -          BEQ WEITER
250  -          STA STORE,X
260  -          INX
270  -          JMP SCHLEIFE1
280  -;
290  -WEITER    STX LAENGE
300  -          LDA #<(TEXT2)
310  -          LDY #>(TEXT2)
320  -          JSR STROUT
330  -          LDA #0
340  -; 0 = AUSGANGSWERT DER PRUEFSUMME
350  -          TAX             ; ZAEHLER = 0
360  -SCHLEIFE2 ROL             ; PRUEFSUMME * 2
370  -          EOR STORE,X
380  -          INX             ; ZAEHLER ERHOEHEN
390  -          CPX LAENGE
400  -          BNE SCHLEIFE2
410  -          CLC
420  -          ADC LAENGE      ; LAENGE ADDIEREN
430  -          TAX             ; PRUEFSUMME
440  -          LDA #0          ; AUSGEBEN
450  -          JSR NUMOUT
460  -          JMP ANFANG      ; NOCH EINMAL
1000 -;
1010 -; TEXTE
1020 -;
1030 -TEXT1     .BY 13
1040 -.TX "----------------------------------------"
1050 -.TX "EINGABE ?  "
1060 -.BY 0
1070 -;
1080 -TEXT2     .BY 13
1090 -.TX "PRUEFSUMME "
1100 -.BY 0
2000 -;
2010 -; ZWISCHENSPEICHER
2020 -;
2030 -LAENGE    .BY 0           ; ZWISCHENSPEICHER
2040 -STORE     .BY 0
2050 -; ↑ AB STORE WIRD DIE EINGABE ABGELEGT
```

Listing 9. Die Berechnung von Prüfsummen

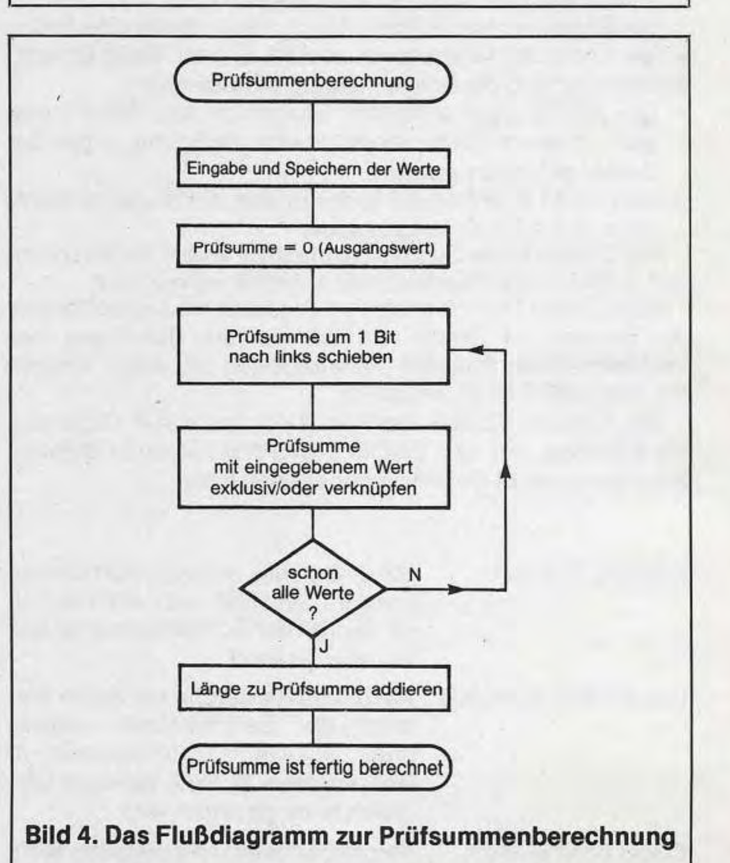


Bild 4. Das Flußdiagramm zur Prüfsummenberechnung

in der die Prüfsumme gefunden wurde. Listing 10 numeriert, im Gegensatz zum SMON die Spalten mit 0 (statt mit 1) beginnend. Außerdem wurde die Adressentabelle in »LOWTAB« (Tabelle der Low-Bytes) und »HIGHTAB« (High-Bytes) zerlegt, was die Programmierung stark erleichtert.

Wir würden zwar Spalten von 1 an numerieren, für den Computer ist es aber besser, mit Spalte 8 zu beginnen. Wenn im X-Register die Spalte (0: VC 20, 1: C 64, 2: PC 128, 3: AMIGA) steht, lesen die Zeilen 620/630 aus einer Tabelle die Adresse, ab der die ASCII-Darstellung des Prozessors zu finden ist. Weil jede der Tabellen »LOWTAB« und »HIGHTAB« gleich viele Elemente wie die Tabelle »PRUEFSUMMEN« hat, muß keine komplizierte Umwandlung über Multiplikation mit 2 oder ähnliches erfolgen wie beispielsweise beim SMON.

Auf eine akute Gefahr bei der Verwendung von Prüfsummen soll jetzt hingewiesen werden: die »Überschneidung von Prüfsummen«:

So wie unterschiedliche Basic-Zeilen beim Checksummer eine gleiche Prüfsumme haben können, sind Prüfsummen nie eindeutig.

Wenn Sie bei Listing 10 etwas herumprobieren, werden Sie sicher feststellen, daß auch eigentlich nicht vorgesehene Eingaben Wirkung zeigen. Dies liegt daran, daß diese Eingaben die gleiche Prüfsumme wie die Taste »VC 20«. »C 64«, »PC 128« oder »AMIGA« haben. Daher sollte man immer darauf achten, daß sich die vorgesehenen Eingaben nicht in ihren Prüfsummen überschneiden (das heißt, die gleichen Prüfsummen haben). Wenn man dies aber beachtet, so ist das Arbeiten mit Prüfsummen, vor allem bei kleineren Datenmengen, eine angenehme Sache.

e) Beispielprogramm für Tabellen

Wenden wir uns jetzt einem etwas größeren (aber keineswegs komplizierteren) Programm zu. Es heißt schlicht und einfach »TABELLEN-BEISPIEL«, womit schon einiges über die Funktion ausgesagt ist: ein reines Beispielprogramm, das nicht den Anspruch erhebt, etwa als Anwendersoftware nützlich zu sein. In Listing 11 finden Sie den kommentierten Quelltext.

Zuerst soll die Bedienung des Programms erläutert werden. Gestartet wird »TABELLEN-BEISPIEL« durch SYS 49152, worauf man sich in folgendem Menü befindet:

```
ZAHL IN ZAHLWORT WANDELN      (0)
BILDSCHIRMFARBE               (1)
RESET AUSLOESEN               (2)
PROGRAMMENDE UEBER RTS        (3)
BITTE AUSWAEHLEN!
```

Die Zahlen in Klammern sehen Sie nicht, diese zeigen nur die interne Numerierung der Menüpunkte an.

Der jeweils angewählte Menüpunkt (unmittelbar nach dem Start: 0) wird im Gegensatz zu den anderen revers hervorgehoben.

Der angewählte Menüpunkt kommt durch Drücken von F1,RETURN, »←« - oder »=«-Taste zur Ausführung.

Wollen Sie einen anderen Menüpunkt anwählen, drücken Sie einfach CRSR DOWN,»D«,F5 oder »+«, um den invertierten Bereich nach unten zu bewegen. Weiter nach oben gelangen Sie über CRSR UP,»U«,F3 oder »–«.

Wenn Sie von »3« aus nach unten wollen, geht es wieder bei »0« los; von »0« nach oben führt auf Punkt »3«.

Auf Punkt »0« (Ausgangseinstellung) kommen Sie über HOME,»0« oder Klammeraffe.

Sicher würden Sie Ihre Programme auch gerne mit einem solch komfortablen Menü aufwerten. Wenn Sie die Beschreibung des Quelltextes gut durchlesen, wird dies keine Schwierigkeiten bereiten.

Nun zu den einzelnen Menüpunkten.

»2« (Reset auslösen) springt in die RESET-Routine ab $FCE2. »3« (Programmende über RTS) bewirkt einen Rücksprung ins Basic. Wenn Sie aber »TABELLEN-BEISPIEL« vom

```
100  -.LI 1,3,0
110  -.BA $C000 ; START: SYS 49152
120  -;
130  -.GL BASIN = $FFCF
140  -.GL NUMOUT = $BDCD
150  -.GL STROUT = $AB1E
160  -;
170  -ANFANG     LDA #<(TEXT1)
180  -           LDY #>(TEXT1)
190  -           JSR STROUT
200  -;
210  -           LDX #0
220  -SCHLEIFE1  JSR BASIN
230  -           CMP #" "         ; SPACE?
240  -           BEQ SCHLEIFE1    ; DANN UEBERLESEN
250  -           CMP #13          ; 13 = RETURN
260  -           BEQ WEITER1
270  -           STA STORE,X
280  -           INX
290  -           JMP SCHLEIFE1
300  -;
310  -WEITER1    STX LAENGE
320  -           LDA #<(TEXT2)
330  -           LDY #>(TEXT2)
340  -           JSR STROUT
350  -           LDA #0
360  -; 0 = AUSGANGSWERT DER PRUEFSUMME
370  -           TAX              ; ZAEHLER = 0
380  -SCHLEIFE2  ROL              ; PRUEFSUMME * 2
390  -           EOR STORE,X
400  -           INX              ; ZAEHLER ERHOEHEN
410  -           CPX LAENGE
420  -           BNE SCHLEIFE2
430  -           CLC
440  -           ADC LAENGE       ; LAENGE ADDIEREN
450  -; HIER STEHT DIE PRUEFSUMME IM AKKU
460  -;
470  -           LDX #0
480  -SCHLEIFE3  CMP PRUEFSUMMEN,X
490  -           BEQ WEITER2
500  -           INX
510  -           CPX #4
520  -           BNE SCHLEIFE3
530  -; PRUEFSUMME NICHT GEFUNDEN
540  -;
550  -           PLA
560  -           PLA
570  -           LDA #<(TEXT3)
580  -           LDY #>(TEXT3)
590  -           JSR STROUT
600  -           JSR ANFANG       ; VON VORNE
610  -;
620  -WEITER2    LDA LOWTAB,X     ; LOW-BYTE
630  -           LDY HIGHTAB,X    ; HIGH-BYTE
640  -           JSR STROUT
650  -           JMP ANFANG       ; NOCH EINMAL!
660  -;
1000 -;
1010 -; TEXTE
1020 -;
1030 -TEXT1      .BY 13
1040 -.TX "--------------------------------------"
1050 -.TX "COMPUTER : "
1060 -.BY 0
1070 -;
1080 -TEXT2      .BY 13
1090 -.TX "PROZESSOR: "
1100 -.BY 0
1110 -;
1120 -TEXT3      .TX "WEISS ICH NICHT!"
1130 -.BY 0
1140 -;
1150 -;
1160 -T6502      .TX "MOS 6502"
1170 -.BY 0
1180 -;
1190 -T6510      .TX "MOS 6510"
1200 -.BY 0
1210 -;
1220 -T8502      .TX "MOS 8502 & Z80"
1230 -.BY 0
1240 -;
1250 -T68000     .TX "MOTOROLA 68000"
1260 -.BY 0
1270 -;
2000 -;
2010 -; NUMERISCHE TABELLEN
2020 -;
2030 -LOWTAB     .BY <(T6502),<(T6510),<(T8502),<(T68000)
2040 -HIGHTAB    .BY >(T6502),>(T6510),>(T8502),>(T68000)
2050 -;
2060 -PRUEFSUMMEN .BY 228,83,149,136
2070 -; REIHENFOLGE: VC20,C64,PC128,AMIGA
3000 -;
3010 -; ZWISCHENSPEICHER
3020 -;
3030 -LAENGE     .BY 0             ; ZWISCHENSPEICHER
3040 -STORE      .BY 0
3050 -; ↑ AB STORE WIRD DIE EINGABE ABGELEGT
```

Listing 10. Eine Anwendung der Prüfsummenberechnung

Hypra-Ass aus gestartet haben, finden Sie sich im »AUTO-NUMBER«-Modus wieder. Dies ist weder ein Fehler von »TABELLEN-BEISPIEL« noch von Hypra-Ass, sondern liegt daran, daß beide Programme eine bestimmte Adresse verwenden, die Hypra-Ass dann als Aufforderung zur automatischen Zeilennumerierung wertet. Am besten starten Sie »TABELLEN-BEISPIEL« nur vom normalen Basic aus.

Punkt »0« bittet Sie um Eingabe einer Zahl von 0 bis 9 und gibt zur eingegebenen Zahl das Zahlwort aus. Beispiel: Eingabe »0«, Ausgabe »NULL«.

Danach müssen Sie eine Taste drücken, um ins Hauptmenü zu kommen.

Punkt »1« schließlich bietet die Möglichkeit, die Hintergrundfarbe besonders elegant einzustellen: Sie geben einfach die Farbe als Wort ein, zum Beispiel SCHWARZ.

Folgende Eingaben sind vorgesehen:
SCHWARZ,WEISS,ROT,TUERKIS,VIOLETT,GRUEN,BLAU,
GELB,ORANGE,BRAUN,HELLROT,GRAU 1,GRAU 2,
HELLGRUEN,HELLBLAU,GRAU 3

Aufgrund der Überschneidung von Prüfsummen zeigen jedoch auch andere Eingaben Wirkung, zum Beispiel:
SCH,HYPRA ASS,PRINT,COMPUTER-GRAPHIK,
TAGESSCHAU

Nun wollen wir uns mit dem Quelltext befassen.

Ab Zeile 10000 finden Sie die Tabellen. Und weil unser Programm ein Beispiel für die Verwendung von Tabellen sein soll, sind es derer recht viele. Die wichtigsten davon sind jedoch analog der internen Numerierung der Menüpunkte aufgebaut, da sie Daten für die Menüsteuerung beinhalten. Diese Tabellen sind auch mit 0 - 3 numeriert und grafisch in Bild 6 dargestellt.

Sehen wir uns wieder den Quelltext, beginnend mit der ersten Zeile, an.

Auf die Symboldefinitionen (210 - 260) folgt die Initialisierung der Hauptschleife (280 - 310). Diese Initialisierung löscht Bildschirm (280) und Tastaturpuffer (290 - 300). Außerdem wird der aktuelle (= derzeit invers dargestellte) Menüpunkt (immer in der Adresse »MPT« enthalten) auf 0 gesetzt (310). Zeile 310 ist also dafür verantwortlich, daß nach dem Start über SYS 49152 das Inversfeld ganz oben steht (auf Punkt 0).

Die Texte, die der Beschreibung der Menüpunkte dienen, werden in der Hauptschleife »HSCHLEIFE« (350 - 550) ausgegeben. Mit dieser wollen wir uns nun eingehend auseinandersetzen.

Zunächst wird die Tabelle »RVSTAB« gelöscht (350 - 400). Diese Tabelle enthält die Information, ob der erläuternde Text zu einem Menüpunkt invers ausgegeben wird. Wenn nein, so enthält das entsprechende Byte eine »0«, andernfalls eine »18« (= REVERS-ON-Code für Betriebssystem). Das entsprechende Byte aus »RVSTAB« braucht nur vor dem Menüpunkt-Text ausgegeben werden (470 - 480). Die Zeilen 410 - 430 sorgen dafür, daß das Byte in »RVSTAB«, welches sich auf den aktuellen Menüpunkt bezieht, den RVS-ON-Code erhält.

In der Hauptschleife muß das X-Register in »XSAVE« gesichert werden, weil die Routine »STROUT« den Inhalt des X-Registers ändert.

Mit »TASTE« (610) beginnt dann die Tastaturabfrage im Menü. Die Routine »GET« holt ein Zeichen von der Tastatur als ASCII-Code in den Akku. Wurde keine Taste gedrückt, erhält der Akku den Code 0. In diesem Fall wartet 620 auf eine neue Eingabe. Beachten Sie bitte, daß der Akku nach der Zeile 620 NIE den Wert 0 haben kann (dies wird sich bald als nützlich erweisen)!

Wurde nun eine Taste gedrückt, sucht »SCHLEIFE« (630 - 680) in der Tabelle »TASTEN«, die im Quelltext ab Zeile 10210 steht, nach dem eingegebenen Zeichen (wird es nicht gefunden, erfolgt in 690 der Sprung zur neuen Eingabe).

```
100  -.BA $C000 ; START: SYS 49152
110  -;
120  -; *************************
130  -; *                       *
140  -; *  TABELLEN - BEISPIEL  *
150  -; *  ===================  *
160  -; *                       *
170  -; *  BY  FLORIAN MUELLER  *
180  -; *                       *
190  -; *************************
200  -;
210  -.GL STROUT = $AB1E
220  -.GL CURSORHOME = $E566
230  -.GL GET = $FFE4
240  -.GL BASIN = $FFCF
250  -.GL BASOUT = $FFD2
260  -.GL RESET = $FCE2 ; SOFTWARE-RESET
270  -;
280  -START      JSR $E544       ; = PRINT CHR$(147)
290  -          LDA #0          ; TASTATURPUFFER
300  -          STA 198         ; LOESCHEN
310  -          STA MPT
320  -; ↑ SETZT AKTUELLEN MENUEPUNKT AUF 0
330  -HSCHLEIFE JSR CURSORHOME
340  -; ↑ HSCHLEIFE = HAUPTSCHLEIFE
350  -          LDA #0
360  -          TAX
370  -SCHLEIFE1 STA RVSTAB,X
380  -          INX
390  -          CPX #4
400  -          BNE SCHLEIFE1
410  -          LDX MPT
420  -          LDA #18         ; 18 = REVERS EIN
430  -          STA RVSTAB,X
440  -          LDX #0
450  -; ↑ SCHLEIFENZAEHLER INITIALISIEREN
460  -SCHLEIFE2 STX XSAVE       ; X RETTEN
470  -          LDA RVSTAB,X
480  -          JSR BASOUT
490  -          LDA TEXTLO,X    ; ERKLAERUNG
500  -          LDY TEXTHI,X    ; ZUM MENUEPUNKT
510  -          JSR STROUT      ; AUSGEBEN
520  -          LDX XSAVE       ; X WIEDER HOLEN
530  -          INX
540  -          CPX #4
550  -          BNE SCHLEIFE2
560  -;
570  -;
580  -; HIER IST DAS MENUE BEREITS AUF
590  -; DEN BILDSCHIRM AUSGEGEBEN WORDEN.
600  -;
610  -TASTE      JSR GET         ; TASTATURABFRAGE
620  -          BEQ TASTE       ; WARTEN AUF TASTENDRUCK
630  -          LDX #0
640  -SCHLEIFE3 CMP TASTEN,X
650  -          BEQ WEITER1
660  -          INX
670  -          CPX #16
680  -          BNE SCHLEIFE3
690  -          JMP TASTE
700  -WEITER1    TXA
710  -          LSR             ; DIVIDIERT AKKU-
720  -          LSR             ; MULATOR DURCH 4
730  -          TAX
740  -          LDA SP1LO,X
750  -          STA SPRUNG
760  -          LDA SP1HI,X
770  -          STA SPRUNG+1
780  -;
790  -.EQ RUECKSPRUNG = HSCHLEIFE-1
800  -; ↑ LEGT RUECKSPRUNGADRESSE DES
810  -;   UNTERPROGRAMMS FEST.
820  -;
830  -          LDA #>(RUECKSPRUNG)
840  -          PHA
850  -          LDA #<(RUECKSPRUNG)
860  -          PHA
870  -          JMP (SPRUNG)
880  -;
890  -;
900  -HOME       LDX #0
910  -          STX MPT
920  -ENDE       RTS             ; ENDE DES UNTERPRG
930  -;
940  -DOWN       LDX MPT         ; MENUEPUNKT
950  -          INX             ; UM 1 ERHOEHEN
960  -          CPX #4          ; GROESSER ALS 3?
970  -          BEQ HOME        ; DANN =0
980  -          STX MPT         ; SONST UEBERNEHMEN
990  -          RTS             ; ZUR HAUPTSCHLEIFE
1000 -;
1010 -UP         LDX MPT         ; MENUEPUNKT
1020 -          DEX             ; DEKREMENTIEREN
1030 -          BPL ENDUP       ; > 0?
1040 -          LDX #3          ; NEIN, DANN =3
1050 -ENDUP      STX MPT         ; UND UEBERNEHMEN
1060 -          RTS             ; ZUR HAUPTSCHLEIFE
1070 -;
1080 -;
1090 -EXEC       PLA             ; STAPELMANIPULATION
1100 -          PLA
1110 -          LDX MPT
```

Listing 11. »Tabellen-Beispiel«, ein Beispiel zur Verwendung von Tabellen

```
1120 -              LDA SP2LO,X
1130 -              STA SPRUNG
1140 -              LDA SP2HI,X
1150 -              STA SPRUNG+1
1160 -              JMP (SPRUNG)
1170 -;
1180 -;
1190 -;
1200 -ZAHLWORT      LDA #<(TZAHL)   ; AUFFORDERUNG
1210 -              LDY #>(TZAHL)   ; ZUR EINGABE
1220 -              JSR STROUT      ; AUSGEBEN
1230 -              JSR BASIN       ; HOLT ZEICHEN
1240 -              SEC             ; IN BINAERZAHL
1250 -              SBC #"0"        ; UMWANDELN
1260 -              TAX             ; INS X-REGISTER
1270 -;
1280 -; JETZT STEHT IM X-REGISTER
1290 -; DIE EINGEGEBENE ZAHL
1300 -;
1310 -              CMP #10         ; > 10?
1320 -              BCC ZAHLWORT1   ; NEIN=> WEITER
1330 -              JMP ZAHLWORT    ; NEUEINGABE
1340 -;
1350 -ZAHLWORT1     STX XSAVE       ; X RETTEN
1360 -              LDA #<(TWORT)   ; AUFFORDERUNG
1370 -              LDY #>(TWORT)   ; ZUR EINGABE
1380 -              JSR STROUT      ; AUSGEBEN
1390 -              LDX XSAVE       ; X WIEDER HOLEN
1400 -              LDA ZWLO,X      ; ADRESSE DES
1410 -              LDY ZWHI,X      ; ZAHLWORTES HOLEN
1420 -              JSR STROUT      ; UND Z.WORT DRUCKEN
1430 -;
1440 -WAIT          JSR GET         ; WARTET AUF
1450 -              BEQ WAIT        ; TASTENDRUCK
1460 -              JMP START       ; ZUM HAUPTMENUE
1470 -;
1480 -;
1490 -;
1500 -FARBE         LDA #<(TFARBE)
1510 -              LDY #>(TFARBE)
1520 -              JSR STROUT
1530 -              LDX #0
1540 -FARBE1        JSR BASIN       ; HOLT EINGABE
1550 -              CMP #" "        ; SPACE ?
1560 -              BEQ FARBE1      ; JA=>UEBERLESEN
1570 -              CMP #13         ; ENDE DER EINGABE?
1580 -              BEQ FARBE2      ; JA, DANN WEITER
1590 -              STA FARBWORT,X  ; EINGABE SPEICHERN
1600 -              INX             ; ZAEHLER ERHOEHEN
1610 -              JMP FARBE1      ; ZUR SCHLEIFE
1620 -FARBE2        STX 2           ; LAENGE MERKEN
1630 -              LDX #0
1640 -              TXA
1650 -FARBE3        ROL
1660 -              EOR FARBWORT,X
1670 -              INX
1680 -              CPX 2           ; SCHON FERTIG?
1690 -              BNE FARBE3      ; NEIN,ZUR SCHLEIFE
1700 -              CLC             ; LAENGE
1710 -              ADC 2           ; ADDIEREN
1720 -;
1730 -; HIER STEHT IM AKKU DIE PRUEFSUMME
1740 -;
1750 -              LDX #0
1760 -FARBE4        CMP PRUEFSUMMEN,X
1770 -              BEQ FARBE5      ; GEFUNDEN
1780 -              INX
1790 -              CPX #16
1800 -              BNE FARBE4
1810 -              JMP FARBE       ; NEUE EINGABE
1820 -FARBE5        STX 53280       ; BILDSCHIRM-
1830 -              STX 53281       ; FARBE SETZEN
1840 -              JMP START       ; ZUM MENUE
1850 -;
10000-;
10010-; TABELLEN
10020-; ========
10030-;
10040-; TEXTE:
10050-;
10060-PUNKT0        .TX "ZAHL IN ZAHLWORT UMWANDELN"
10070-.BY 13,13,0
10080-;
10090-PUNKT1        .TX "BILDSCHIRMFARBE"
10100-.BY 13,13,0
10110-;
10120-PUNKT2        .TX "RESET AUSLOESEN"
10130-.BY 13,13,0
10140-;
10150-PUNKT3        .TX "PROGRAMMENDE UEBER RTS"
10160-.BY 13,13,13
10170-.TX "BITTE AUSWAEHLEN !"
10180-.BY 0
10190-;
10200-;
10210-TASTEN        .BY 133,13,"←","="; 133=F1,13=RETURN
10220-.BY 19,"0","@",0 ; 19=HOME,0=DUMMY
10230-.BY 17,"D",135,"+" ; 17=CRSR DOWN,135=F5
10240-.BY 145,"U",134,"-" ; 145=CRSR UP,134=F3
10250-;
10260-;
10270-TZAHL         .BY 147         ; CLEAR HOME
10280-.TX "ZAHL (0-9) ? "
10290-.BY 0
10300-;
10310-TWORT         .TX " IN WORTEN : "
10320-.BY 0
10330-;
10340-;
10350-; ZAHLWOERTER (0-9)
10360-;
10370-;
10380-NULL          .TX "NULL"
10390-.BY 0
10400-;
10410-EINS          .TX "EINS"
10420-.BY 0
10430-;
10440-ZWEI          .TX "ZWEI"
10450-.BY 0
10460-;
10470-DREI          .TX "DREI"
10480-.BY 0
10490-;
10500-VIER          .TX "VIER"
10510-.BY 0
10520-;
10530-FUENF         .TX "FUENF"
10540-.BY 0
10550-;
10560-SECHS         .TX "SECHS"
10570-.BY 0
10580-;
10590-SIEBEN        .TX "SIEBEN"
10600-.BY 0
10610-;
10620-ACHT          .TX "ACHT"
10630-.BY 0
10640-;
10650-NEUN          .TX "NEUN"
10660-.BY 0
10670-;
10680-;
10690-TFARBE        .BY 147         ; CLEAR HOME
10700-.TX "WELCHE FARBE ? "
10710-.BY 0
10720-;
10730-;
10740-RVSTAB        .BY 0,0,0,0     ; 4 BYTES RESERVIEREN
10750-;
10760-;
10770-; ZAHLEN:
10780-;
10790-; ADRESSEN DER TEXTE, DIE DIE
10800-; MENUEPUNKTE BESCHREIBEN
10810-;
10820-TEXTLO        .BY <(PUNKT0),<(PUNKT1)
10830-.BY <(PUNKT2),<(PUNKT3)
10840-;
10850-TEXTHI        .BY >(PUNKT0),>(PUNKT1)
10860-.BY >(PUNKT2),>(PUNKT3)
10870-;
10880-;
10890-; ADRESSEN DER ZAHLWOERTER
10900-;
10910-ZWLO          .BY <(NULL),<(EINS),<(ZWEI),<(DREI)
10920-.BY <(VIER),<(FUENF),<(SECHS),<(SIEBEN)
10930-.BY <(ACHT),<(NEUN)
10940-;
10950-ZWHI          .BY >(NULL),>(EINS),>(ZWEI),>(DREI)
10960-.BY >(VIER),>(FUENF),>(SECHS),>(SIEBEN)
10970-.BY >(ACHT),>(NEUN)
10980-;
10990-;
11000-; ADRESSEN DER UNTERROUTINEN
11010-; FUER DIE MENUESTEUERUNG
11020-;
11030-SP1LO         .BY <(EXEC),<(HOME),<(DOWN),<(UP)
11040-;
11050-SP1HI         .BY >(EXEC),>(HOME),>(DOWN),>(UP)
11060-;
11070-;
11080-; ADRESSEN DER EINZELNEN
11090-; MENUEPUNKTE
11100-;
11110-SP2LO         .BY <(ZAHLWORT),<(FARBE)
11120-.BY <(RESET),<(ENDE) ; BEI ENDE STEHT
11130-SP2HI         .BY >(ZAHLWORT),>(FARBE)
11140-.BY >(RESET),>(ENDE) ; EIN RTS-BEFEHL
11150-;
11160-; PRUEFSUMMEN DER FARB-WOERTER
11170-;
11180-PRUEFSUMMEN .BY 41,158,137,212,159,101
11190-.BY 3,2,33,69,201,116,113,121,127,114
11200-;
11210-;
11220-; ZWISCHENSPEICHER
11230-;
11240-MPT           .BY 0           ; 1 BYTE RESERVIEREN
11250-XSAVE         .BY 0
11260-SPRUNG        .WO 0           ; 2 BYTES FREIHALTEN
11270-FARBWORT      .BY 0
11280-; ↑ AB 'FARBWORT' WIRD DIE EINGABE
11290-;   DER FARB-BEZEICHNUNG ABGELEGT.

READY.
```

Spalte	0	1	2	3	
TEXTLO	0	1	2	3	Adressen der Text-Tabellen, die die Menüpunkte beschreiben
TEXTHI	0	1	2	3	
zeigen als Vektoren auf:	PUNKT 0	PUNKT 1	PUNKT 2	PUNKT 3	Text-Tabellen
RVSTAB	0	1	2	3	enthält Information, ob Text zu Menüpunkt invertiert werden soll
SP2LO	0	1	2	3	enthält die Adressen der Routinen zu den Menüpunkten
SP2HI	0	1	2	3	
zeigen als Vektoren auf:	Routine ZAHLWORT	Routine FARBE	Routine RESET	Routine ENDE	

Bild 5. So verwendet man Tabellen zur Realisierung eines Menüs

Diese Tabelle »TASTEN« enthält alle vorgesehenen Tastendrücke zur Menüsteuerung, die in 4er-Blockweise angeordnet sind (Bild 5). Nach der Suchschleife steht im X-Register die Position der gedrückten Taste innerhalb der Tabelle »TASTEN« (zum Beispiel 0 = F1 gedrückt, 4 = HOME gedrückt). Diese Position wird – ohne Berücksichtigung des Divisions-Restes – durch 4 dividiert (700 - 730), um festzuhalten, von welchem Tastenblock eine Taste gedrückt wurde.

Dadurch ist eindeutig bestimmt, welche Befehlsgruppe aufgerufen werden muß.

Steht nach 730 im X-Register 0, wurde eine der ersten vier in »TASTEN« enthaltenen Tasten gedrückt, die die Ausführung des aktuellen Menüpunktes veranlassen (Zeile 10210 und Bild 5). Ist X=1, so wurde eine Taste aus Zeile 10220 gedrückt. In 10220 steht als letztes Byte eine 0. Diese dient, da für die Funktion »Inversfeld in HOME-Position« nur drei Tastendrücke vorgesehen wurden, zum Auffüllen auf vier Tasten. 0 kann hier bedenkenlos als Dummy (Füllbyte ohne wirkliche Bedeutung) stehen, da der Akku aufgrund von 620 nie den Wert 0 annehmen wird.

Beinhaltet X nach der Division durch 4 den Wert 2, wird das Inversfeld nach unten bewegt, ist X=3, dann nach oben. Dies können Sie sich an Bild 6 veranschaulichen.

An den Zeilen 740 - 870 sehen wir nun die Verwendung einer Sprungtabelle. Unsere Sprungtabelle ist »SP1LO/SP1HI«. »SP1LO« beinhaltet die Low-, »SP1HI« die High-Bytes der anzuspringenden Routinen. In den Vektor »SPRUNG« wird einfach die Zieladresse geschrieben (740 - 770).

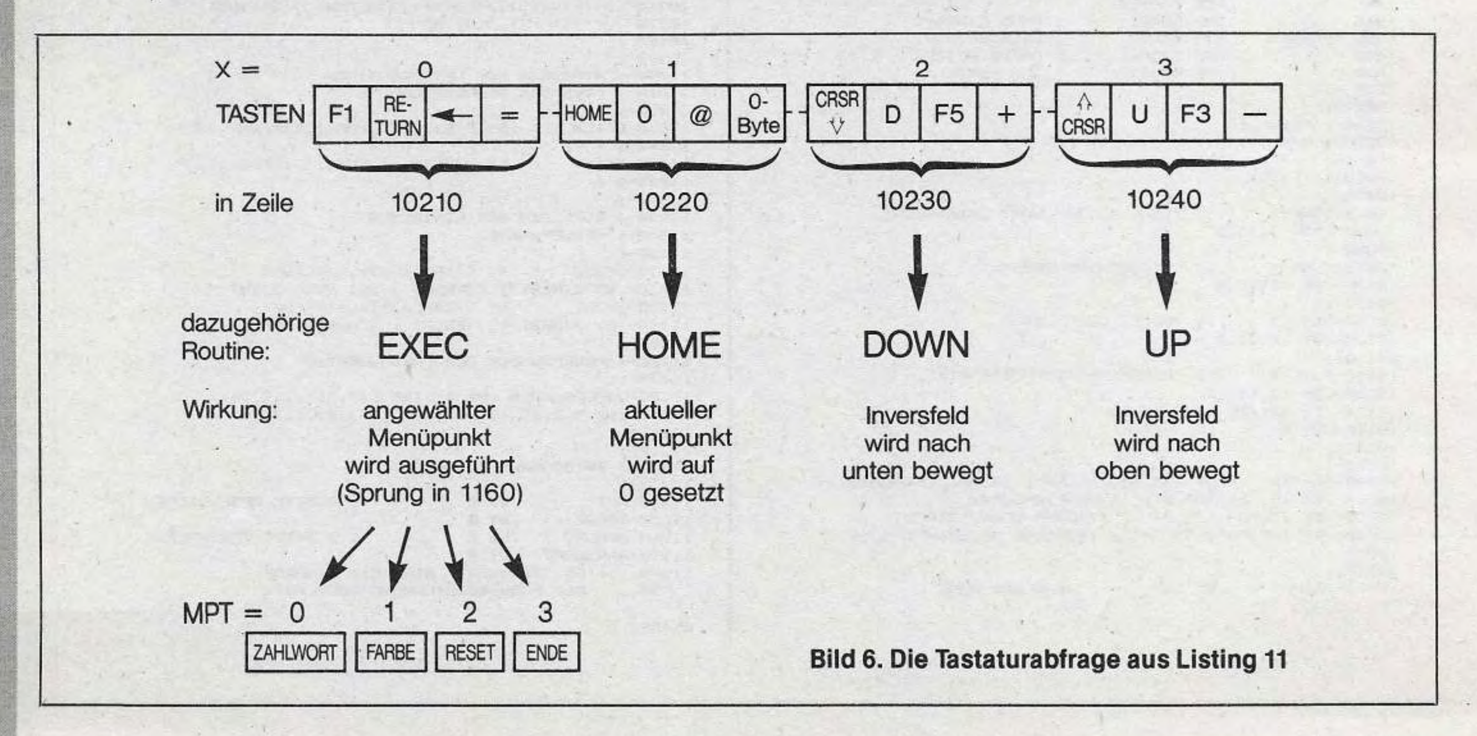


Bild 6. Die Tastaturabfrage aus Listing 11

Die Zuweisungszeile 790 errechnet die Rücksprungadresse des aufzurufenden Unterprogramms. Bei einem RTS soll nämlich zur »HSCHLEIFE« gesprungen werden.

Diese Rücksprungadresse »RUECKSPRUNG« wird auf den Stapel gelegt (830 - 860), zuletzt erfolgt der indirekte Sprung (870). Die über die soeben beschriebene Simulation von JSR (ind) angesprungenen Routinen finden Sie ab Zeile 900. Es wird einfach der aktuelle Menüpunkt »MPT« entsprechend dem Tastendruck geändert, dann wird zur »HSCHLEIFE« gesprungen, die auch die Tabelle »RVSTAB« entsprechend anpaßt.

»EXEC« (1090) holt die Rücksprungadresse vom Stapel (1090 - 1100), da diese Routine nicht als Unterprogramm behandelt werden soll.

Die Zeile 1110 holt den angeforderten Menüpunkt ins X-Register. Dann wird aus »SP2LO/SP2HI« die Adresse der zum Menüpunkt gehörenden Routine geholt und diese über einen gewöhnlichen indirekten Sprung aufgerufen (1160).

Als Routine zu »2« wird einfach die RESET-Routine des Betriebssystems angesprungen, für »3« eignet sich jeder RTS-Befehl, also auch der bei »ENDE« (920).

»ZAHLWORT«, die Routine zu 0, holt eine Zahl als ASCII-Code (1230) und wandelt sie in einen numerischen Wert um (1240 - 1250), indem der ASCII-Code von 0 abgezogen wird. Das Ergebnis landet im X-Register (1260). Ob auch eine Zahl eingegeben wurde, prüfen die Zeilen 1310 - 1330. Bei »ZAHLWORT« (1350) wird das Resultat der Subtraktion in »XSAVE« gesichert, der Text »IN WORTEN« ausgegeben und das X-Register wieder geholt.

Die Tabelle »ZWLO/ZWHI« enthält die Adressen, ab denen die Texte der Zahlwörter als ASCII-Code stehen. Aus »ZWLO/ZWHI« wird dann diese Adresse geholt (1400 - 1410) und der dort stehende Text ausgegeben (1420). Danach erwartet das Programm noch einen Tastendruck (1440-1450), bevor ins Hauptmenü verzweigt wird (1460).

Als letzte Routine wird »FARBE« besprochen (1500-1850): Hierzu ist jedoch aufgrund der Ähnlichkeit zu Listing 10 nicht viel zu erläutern. Bei 1820 steht im X-Register der Code der eingegebenen Farbe (= Position der Prüfsumme innerhalb der Tabelle »PRUEFSUMMEN«). Dieser muß nur noch in die entsprechenden VIC-Register geschrieben werden (1820-1830). Ab Zeile 10000 stehen dann die Tabellen. Wenn Sie die Tabellen angesehen haben, sollten Sie durchaus noch einmal den Quelltext bis 10 000 betrachten und die hier endende Beschreibung des Programms lesen. Denn wenn Sie das Programm »TABELLEN-BEISPIEL« ganz verstanden haben, sind Sie einen großen Schritt in der Assemblerprogrammierung weitergekommen!

Ich könnte mir übrigens vorstellen, daß Sie in Ihren eigenen Programmen jetzt auch eine Menüsteuerung wie die in »TABELLEN-BEISPIEL« einbauen; wie das geht, können Sie dem Programm »TABELLEN-BEISPIEL« entnehmen.

Eine Anmerkung ist wichtig: »TABELLEN-BEISPIEL« kann noch weiter verbessert werden. Sie werden sehen, daß viele Stellen noch optimiert werden können. Insbesondere der Speicherplatzbedarf kann verringert werden.

f) Weitere Anregungen zur Anwendung von Tabellen

Auch die bisherigen Erläuterungen und das Beispielprogramm können die Kreativität des Programmierers nicht ersetzen, sondern nur die Programmierung erleichtern. Aus diesem Grund möchte ich Ihnen noch einige Beispiele nennen, wie sich Tabellen sinnvoll verwerten lassen.

- Ein Anwenderprogramm, das aus Menüs und Untermenüs besteht, sollte in einer Tabelle die Adressen der Menüs/Untermenüs speichern.
- Spiele müssen oft viele Spritebewegungen, die immer gleich sind, durchführen. Es empfiehlt sich, die Spritebewegungen als Koordinaten in einer Tabelle abzulegen.
- Bei Software-Interfaces müssen viele Umrechnungen erfolgen. Durch eine Umwandlungstabelle können diese stark beschleunigt werden.
- Naturwissenschaftlich orientierte Programme müssen verschiedene Maße umrechnen. Die Umrechnungswerte können in einer Tabelle untergebracht werden.

Dies soll nur eine Anregung sein. Ich wüßte aber kein komplexes Programm, das sich nicht durch den gezielten Einsatz von Tabellen vereinfachen und beschleunigen ließe.

Texteinschub #1: Fließkommazahlen

Im Text wurde ein Verfahren vorgestellt, um eine Zahl ins MFLPT-Format (MELPT=Memory floating point) umzuwandeln. Das 5 Byte lange Ergebnis dieser Umwandlung kann man dann als KONSTANTE handhaben. Konstanten sind feste, vorausberechnete Werte, die man mit Hilfe der Routine »MEMFAC« in den FAC (Flieskomma-AKKU) kopieren kann. Für viele Werte ist es jedoch überflüssig, die Umwandlung durchzuführen und eine entsprechende Tabelle anzulegen, da sie schon im ROM vorhanden sind. Im Kurs »Assembler ist keine Alchimie« wurden solche Konstanten mitsamt ihrer Adressen schon in einer Tabelle vorgestellt.

Um mit Konstanten (für die Rechenroutinen macht es keinen Unterschied, ob diese im RAM oder im ROM stehen) zu rechnen, kann man diese wie gesagt, in den FAC kopieren und alle weiteren Operationen auf diesen beziehen. Dies war in Listing 8 bei den Funktionen SQR und LOGNAT ausreichend.

Oft möchte man aber den Inhalt des FAC nicht mit einer Funktion wie SQR behandeln, sondern mit anderen Konstanten addieren, multiplizieren und so weiter.

Dafür möchte ich Ihnen im folgenden weitere Interpreter-Routinen vorstellen, die das Rechnen mit Konstanten ermöglichen. Da fast immer in den Akku das Low-, ins Y-Register das High-Byte der Adresse, ab der die Konstante abgelegt ist, geladen werden muß, definieren wir noch vorher folgende Makro:

```
-.MA LDAY (ADRESSE)
-           LDA #<(ADRESSE)
-           LDY #>(ADRESSE)
-.RT
```

Nun zu den Routinen, bei deren Parameterübergabe wir uns auf das Makro LDAY stützen wollen:

ADDMEM	FAC+Konstante → FAC	...LDAY (KONSTANTE) JSR $B867
ADD0,5	FAC+0.5 → FAC	JSR $B849
SUBMEM	Konstante-FAC → FAC	...LDAY (KONSTANTE) JSR $B850
MULMEM	Konstante*FAC → FAC	...LDAY (KONSTANTE) JSR $BA28
MULT10	FAC*10 → FAC	JSR $BAE2
DIVMEM	Konstante/FAC → FAC	...LDAY (KONSTANTE) JSR $BB0F
DIVS10	FAC/10 → FAC	JSR $BAFE
CMPMEM	vergleicht Konstante mit FAC FAC < Konstante: Akku=$FF FAC=Konstante: Akku=$00 FAC > Konstante: Akku=$01	...LDAY (KONSTANTE) JSR $BC5B
POTMEM	Konstante ↑ FAC → FAC	...LDAY (KONSTANTE) JSR $BF78
POTE	e ↑ FAC → FAC	JSR $BFED
MEMFAC	holt Konstante in FAC	...LDAY (KONSTANTE) JSR $BBA2
FACMEM	FAC ab Konstante als MFLPT-Zahl ablegen	LDX # < (KONSTANTE) LDY # > (KONSTANTE) JSR $BBD7
FACOUT	gibt FAC aus	JSR $AABC

7. Die Initialisierung

»Initialisierung« nennt man eine Routine, die vor einem Programmteil (meist einer Schleife) steht und diese vorbereitet. Die Initialisierung wird nur einmal, eine Schleife aber mehrfach durchlaufen. Deshalb bringt es einen Geschwindigkeitszuwachs, wenn die Initialisierung der Schleife Arbeit abnimmt.

Ein Beispiel: Wenn ein Basic-Programm mit »RUN« gestartet wird, werden alle Variablen gelöscht, Files geschlossen und die Adressen, ab denen die Variablen abgelegt werden dürfen, errechnet. Dies ist die Initialisierung der Interpreterschleife. Dann wird Byte für Byte des Basic-Programms eingelesen und bearbeitet.

Muß im gerade übersetzten Befehl ein Sprung (GOTO 500 oder ähnliches) durchgeführt werden, kostet dies bekanntlich viel Zeit, wenn das Sprungziel am Ende eines langen Programms steht. Dies ist darauf zurückzuführen, daß der Interpreter, beginnend mit der ersten Zeile, das ganze Programm nach der Sprungzeile durchsucht, bis er sie gefunden hat.

Diese Berechnung der Adressen wird bei jedem »GOTO« oder »GOSUB« neu durchgeführt.

Viel besser und schneller wäre folgende Vorgehensweise: Bei »RUN« wird zunächst eine Tabelle angelegt, in der die Adressen aller Zeilen enthalten sind. Diese Tabelle könnte zum Beispiel als Array definiert werden. Folgt nun ein Sprung, kann aus der Tabelle die Adresse der Zeile im Speicher geholt werden.

Damit haben wir noch ein wesentliches Merkmal der Initialisierungsroutinen gefunden: Die Initialisierung kann Tabellen anlegen, die dann von der Hauptschleife verarbeitet werden.

Aber nicht nur Tabellen können generiert werden, auch die Berechnung von Flags ist sinnvoll. So merkt sich die »LOAD/VERIFY«-Routine ($FFD5), ob ein Verifizieren oder Laden gewünscht wird. Die Ladeschleife liest dann ein Zeichen von der Floppy oder der Datasette ein und entscheidet erst anschließend, ob das Byte im Speicher abgelegt oder mit dem Speicher verglichen werden soll.

Halten wir also fest, daß Initialisierungsroutinen Schleifen entlasten können. Näher werden wir uns damit beim Thema »Schleifen« beschäftigen.

8. Die Nutzung der Zeropage

In jedem Assembler-Lehrbuch werden die Vorteile der Zeropage-Adressierung gepriesen. Speicherplatzersparnis und hohe Verarbeitungsgeschwindigkeit sind nicht die einzigen Vorzüge; die indirekt-indizierte Adressierung kann nur auf Zeropage-Adressen zugreifen, nicht auf absolute 16-Bit-Adressen. Damit wird der Leser aber schon alleine gelassen. Er erfährt nicht, welche Adressen in der Zeropage für die Praxis geeignet sind. Das wird nun nachgeholt.

Fast die ganze Zeropage wird durch Basic-Interpreter und Betriebssystem belegt. Deshalb führen bestimmte Werte in Zeropage-Adressen oft zum Absturz oder sonstigem Fehlverhalten des Computers.

Wie dies im einzelnen aussieht, erfahren Sie in der Serie »Memory Map mit Wandervorschlägen«, die im 64'er Stammheft erscheint. Nicht nur in Zweifelsfällen stellt diese Serie das optimale Nachschlagewerk dar.

Ich möchte Ihnen nun zeigen, welche Adressen Sie als (Zwischen-)Speicher ohne Schwierigkeiten verwenden können, beziehungsweise was Sie bei Verwendung von Zeropage-Adressen beachten müssen.

a) Adressen, die problemlos verwendet werden können

Auf die Adressen $02 und $FB – $FE wird weder vom

```
,6000  A2 00     LDX #00
,6002  B5 02     LDA   02,X
,6004  9D 00 6F  STA 6F00,X
,6007  E8        INX
,6008  E0 FE     CPX #FE
,600A  D0 F6     BNE 6002
```
Listing 12

```
,6000  A2 FE     LDX #FE
,6002  B5 01     LDA #01
,6004  9D FF 6E  STA 6EFF,X
,6007  CA        DEX
,6008  D0 F8     BNE 6002
```
Listing 13

```
,6000  A2 FE     LDX #FE
,6002  BD FF 6E  LDA 6EFF,X
,6005  95 01     STA #01
,6007  CA        DEX
,6008  D0 F8     BNE 6002
```
Listing 14

```
,6000  A2 34     LDX #34
,6002  B5 16     LDA   16,X
,6004  9D 00 6F  STA 6F00,X
,6007  CA        DEX
,6008  10 F8     BPL 6002
```
Listing 15

```
,6000  A2 34     LDX #34
,6002  BD 00 6F  LDA 6F00,X
,6005  95 16     STA   16,X
,6007  CA        DEX
,6008  10 F8     BPL 6002
```
Listing 16

```
,6000  A2 FF     LDX #FF
,6002  BD 00 01  LDA 0100,X
,6005  9D 00 6F  STA 6F00,X
,6008  CA        DEX
,6009  D0 F7     BNE 6002
,600B  AD 00 01  LDA 0100
,600E  8D 00 6F  STA 6F00
,6011  BA        TSX
,6012  8E 00 70  STX 7000
```
Listing 17

```
,6000  A2 FF     LDX #FF
,6002  BD 00 6F  LDA 6F00,X
,6005  9D 00 01  STA 0100,X
,6008  CA        DEX
,6009  D0 F7     BNE 6002
,600B  AD 00 6F  LDA 6F00
,600E  8D 00 01  STA 0100
,6011  AE 00 70  LDX 7000
,6014  9A        TXS
```
Listing 18

Basic-Interpreter noch vom Betriebssystem zugegriffen. Lediglich bei Initialisierung der Arbeitsspeicher (RESET) werden Sie auf 0 gesetzt.

Für die Praxis heißt das, daß Ihnen die genannten Adressen völlig zur Verfügung stehen.

b) Adressen, die in keiner Weise verwendet werden sollten

Von anderen Adressen hingegen müssen wir unsere Finger lassen. Diese haben entweder elementare Funktionen für Betriebssystem oder CPU, oder werden von beiden dauernd geändert, so daß die Datensicherheit in Frage gestellt ist. Genauer soll hier nicht unterschieden werden.

Belassen Sie die Adressen $00 und $01 unverändert, da sie (siehe Memory Map) für die CPU wichtige Informationen beinhalten und außerdem einige Bits nur durch externe Vorgänge geändert werden.

Das Betriebssystem und der Basic-Interpreter beanspruchen alle bislang ungenannten Adressen.

Von Bildschirmeditor und Tastaturabfrage werden die Adressen $C6 - $F6 beeinflußt. Die Adressen $90 - $C2 dienen der Ein-/Ausgabe-Steuerung mit Peripheriegeräten und der Verwaltung offener Files. Einzige Ausnahme: $A0 - $A2 (interne Uhr). Wenn ein Maschinenprogrammm in ein Basic-Programm eingebaut ist, sind die Adressen $03 - $56 sowie $73 - $8B tabu.

c) Bedingt einsetzbar

Der Vektor $C3/$C4 wird durch RUN/STOP-Restore, RESET oder LOAD beeinflußt. Ansonsten kann mit $C3/$C4 frei verfahren werden.

Ganz Vorsichtige können diesen Vektor auf seinen Ausgangswert $FD30 setzen, sobald das Programm die Adressen $C3/$C4 nicht mehr für eigene Zwecke benötigt.

d) Adressen, die unter Verzicht auf Kassettenbetrieb verwendet werden können

Die folgenden Adressen können verwendet werden, wenn nicht auf RS232 oder Datasette zugegriffen wird.

$9E/$9F, $A5-$A7, $A9-$AB, $B0-$B6, $F7-$FA

Bei anderen Adressen, die sich auf den RS232- oder Kassettenbetrieb beziehen, ist Vorsicht angebracht.

e) Geeignete Zwischenspeicher

Die Adressen $22-$2A und $57-$60 sind sogenannte »verschieden genutzte Arbeitsbereiche«. Sie werden vom Basic-Interpreter vor allem bei arithmetischen Operationen als Zwischenspeicher verwendet. Als solche Zwischenspeicher können wir sie auch verwenden. Sobald allerdings bestimmte Interpreterroutinen aufgerufen werden, können die Inhalte dieser Adressen verlorengehen. Eine längerfristige Aufbewahrung von Daten in diesen Adressen ist zwar nicht möglich, andererseits können wir aber durch Schreibzugriffe auf diese Adressen das Betriebssystem oder den Basic-Interpreter nicht stören.

Zu sagen wäre noch, daß die Adressen $57 - $60 den wichtigen Routinen BLTUC und UMULT (siehe »Assembler ist keine Alchimie«) als Zwischenspeicher dienen.

f) Zeropage kopieren

Zum Abschluß dieses Abschnittes über die Nutzung der Zeropage möchte ich Ihnen noch einen kleinen Trick verraten, der von einigen professionellen Programmen angewandt wird.

Wir sichern die Zeropage-Inhalte in einem anderen Bereich, zum Beispiel von $6F00 an.

Dann können wir viele Adressen in der Zeropage nutzen, sofern wir keine Interpreter- oder Betriebssystemroutine aufrufen. Danach schreiben wir die Zeropage wieder von der Kopie, zum Beispiel von $6F00, zurück und können wie gewöhnlich fortfahren.

Die Adressen 0 und 1 kopieren wir nicht, weil diese nach wie vor für solche Zwecke nutzlos sind. Ebenso könnten wir

```
,6000  A9 D2     LDA #D2
,6002  85 14     STA   14
,6004  A9 3F     LDA #3F
,6006  85 15     STA   15
,6008  A0 00     LDY #00
,600A  B1 14     LDA  (14),Y
,600C  49 FF     EOR #FF
,600E  91 14     STA  (14),Y
,6010  E6 14     INC   14
,6012  D0 02     BNE 6016
,6014  E6 15     INC   15
,6016  A5 14     LDA   14
,6018  C9 60     CMP #60
,601A  A5 15     LDA   15
,601C  E9 47     SBC #47
,601E  90 EA     BCC 600A
```

Listing 19

```
,6000  A9 5F     LDA #5F
,6002  85 14     STA   14
,6004  A9 47     LDA #47
,6006  85 15     STA   15
,6008  A0 00     LDY #00
,600A  B1 14     LDA  (14),Y
,600C  49 FF     EOR #FF
,600E  91 14     STA  (14),Y
,6010  A5 14     LDA   14
,6012  D0 02     BNE 6016
,6014  C6 15     DEC   15
,6016  C6 14     DEC   14
,6018  A5 14     LDA   14
,601A  C9 D2     CMP #D2
,601C  A5 15     LDA   15
,601E  E9 3F     SBC #3F
,6020  B0 E8     BCS 600A
```

Listing 20

```
,6000  A9 00     LDA #00
,6002  85 14     STA   14
,6004  A9 20     LDA #20
,6006  85 15     STA   15
,6008  A0 00     LDY #00
,600A  B1 14     LDA  (14),Y
,600C  49 FF     EOR #FF
,600E  91 14     STA  (14),Y
,6010  C8        INY
,6011  D0 F7     BNE 600A
,6013  E6 15     INC   15
,6015  A5 15     LDA   15
,6017  C9 40     CMP #40
,6019  D0 EF     BNE 600A
```

Listing 21

```
,6000  A9 00   LDA #00
,6002  85 14   STA  14
,6004  A8      TAY
,6005  A9 20   LDA #20
,6007  85 15   STA  15
,6009  AA      TAX
,600A  B1 14   LDA  (14),Y
,600C  49 FF   EOR #FF
,600E  91 14   STA  (14),Y
,6010  C8      INY
,6011  D0 F7   BNE 600A
,6013  E6 15   INC  15
,6015  CA      DEX
,6016  D0 F2   BNE 600A
```

Listing 22

nur einzelne Bereiche kopieren (zum Beispiel die Zeiger für Basic-Programme $16 - $4A). Dann dürfen wir aber auch nur diesen Bereich verändern.

Wenn wir nun den Bereich $02 - $FF kopieren, stehen uns folgende Adressen zur Verfügung:

$03-$06, $14-$86, $71-$8A, $C3/$C4, $FB-$FF

Diese Adressen können Sie nur so lange verwenden, bis eine Routine des Betriebssystems oder Basic-Interpreters aufgerufen wird. Davor muß die alte Zeropage zurückgeschrieben werden.

Da Sie auf diese Weise viel Speicherplatz in der Zeropage gewonnen haben, ist es sogar möglich, eine Tabelle aus Geschwindigkeitsgründen in die Zeropage zu verlegen. Damit steigt auch der Wert der indiziert-indirekten Adressierung erheblich.

Dennoch ist der Speicherplatz in der Zeropage begrenzt. Überlegen Sie sich also, auf welche Werte besonders schnell zugegriffen werden muß und schreiben Sie vorzugsweise diese in die Zeropage.

```
70   -.BA $C000
80   -.LI 1,3,0
90   -;
100  -; ******************************
110  -; *     QUELLTEXTE (HYPRA-ASS)    *
120  -; *     ======================    *
130  -; *                              *
140  -; * FUER VERSCHIEDENE SCHLEIFEN *
150  -; *                              *
160  -; * 28.08.85 BY FLORIAN MUELLER *
170  -; *                              *
180  -; ******************************
190  -;
200  -;
210  -; QUELLTEXT ZU LISTING 1
220  -; =====================
230  -;
240  -.EQ ANFANGSADRESSE = $02
250  -.EQ ENDADRESSE = $FF
260  -.EQ ZIELBEREICH = $6F00
270  -;
280  -          LDX #0
290  -SCHLEIFE1 LDA ANFANGSADRESSE,X
300  -          STA ZIELBEREICH,X
310  -          INX
320  -          CPX #(ENDADRESSE+1-ANFANGSADRESSE)
330  -          BNE SCHLEIFE1
340  -;
350  -;
360  -; QUELLTEXT ZU LISTING 2
370  -; =====================
380  -;
390  -.EQ ANFANGSADRESSE = $02
400  -.EQ ENDADRESSE = $FF
410  -.EQ ZIELBEREICH = $6F00
420  -;
430  -          LDX #(ENDADRESSE+1-ANFANGSADRESSE)
440  -SCHLEIFE2 LDA ANFANGSADRESSE-1,X
450  -          STA ZIELBEREICH-1,X
460  -          DEX          ; DEKREMENTIERBEFEHL
470  -          BNE SCHLEIFE2
480  -;
490  -;
500  -; QUELLTEXT ZU LISTING 4
510  -; =====================
520  -;
530  -.EQ ANFANGSADRESSE = $16
540  -.EQ ENDADRESSE = $4A
550  -.EQ ZIELBEREICH = $6F00
560  -;
570  -          LDX #(ENDADRESSE-ANFANGSADRESSE)
580  -SCHLEIFE3 LDA ANFANGSADRESSE,X
590  -          STA ZIELBEREICH,X
600  -          DEX
610  -          BPL SCHLEIFE3  ; PRUEFT N-FLAG
620  -;
630  -;
640  -; QUELLTEXT ZU LISTING 8
650  -; =====================
660  -;
670  -.EQ ANFANGSADRESSE = $3FD2
680  -.EQ ENDADRESSE = $475F
690  -.EQ ZAEHLER = $14
700  -;
710  -          LDA #<(ANFANGSADRESSE)
720  -          STA ZAEHLER
730  -          LDA #>(ANFANGSADRESSE)
740  -          STA ZAEHLER+1
750  -          LDY #0
760  -SCHLEIFE4 LDA (ZAEHLER),Y
770  -          EOR #$FF
780  -          STA (ZAEHLER),Y
790  -          INC ZAEHLER
800  -          BNE WEITER
810  -          INC ZAEHLER+1
820  -WEITER    LDA ZAEHLER
830  -          CMP #<(ENDADRESSE+1)
840  -          LDA ZAEHLER+1
850  -          SBC #>(ENDADRESSE+1)
860  -          BCC SCHLEIFE4
870  -;
880  -;
890  -; QUELLTEXT ZU LISTING 10
900  -; =====================
910  -;
920  -.EQ ANFANGSADRESSE = $2000
930  -.EQ ENDADRESSE   = $3FFF
940  -.EQ ZAEHLER = $14
950  -;
960  -          LDA #<(ANFANGSADRESSE)
970  -          STA ZAEHLER
980  -          LDA #>(ANFANGSADRESSE)
990  -          STA ZAEHLER+1
1000 -          LDY #0
1010 -SCHLEIFE5 LDA (ZAEHLER),Y
1020 -          EOR #$FF
1030 -          STA (ZAEHLER),Y
1040 -          INY
1050 -          BNE SCHLEIFE5
1060 -          INC ZAEHLER+1
1070 -          LDA ZAEHLER+1
1080 -          CMP #>(ENDADRESSE+1)
1090 -          BNE SCHLEIFE5
1100 -;
1110 -;
1120 -; QUELLTEXT ZU EINER SCHLEIFE,
1130 -; DIE DEN BEREICH $3FD2-$47D1
1140 -; KOMPLEMENTIERT
1150 -;
1160 -.EQ ANFANGSADRESSE = $3FD2
1170 -.EQ ENDADRESSE = $47D1
1180 -.EQ ZAEHLER = $14
1190 -;
1200 -          LDA #<(ANFANGSADRESSE)
1210 -          STA ZAEHLER
1220 -          LDA #>(ANFANGSADRESSE)
1230 -          STA ZAEHLER+1
1240 -          LDX #>(ENDADRESSE+1-ANFANGSADRESSE)
1250 -          LDY #0
1260 -SCHLEIFE6 LDA (ZAEHLER),Y
1270 -          EOR #$FF
1280 -          STA (ZAEHLER),Y
1290 -          INY
1300 -          BNE SCHLEIFE6
1310 -          INC ZAEHLER+1
1320 -          DEX
1330 -          BNE SCHLEIFE6
1340 -;
1350 -; ENDE VON LISTING 12
```

Listing 23

9. Schleifenprogrammierung

Zunächst befassen wir uns mit Schleifen, die maximal 256mal durchlaufen werden.

Typ a: Schleifen mit maximal 256 Durchläufen

Da 256 verschiedene Zahlen mit einem 8-Bit-Prozessor dargestellt werden können, verwendet man hier das X- (oder Y-) Register als Schleifenzähler. In Listing 12 sehen Sie die einfachste Form einer Schleife, die die Zeropage-Adressen $02 - $FF nach $6F00 kopiert.

Da der Schleifenzähler X in Listing 12 INKREMENTIERT wird, haben wir es mit einer INKREMENTIERSCHLEIFE zu tun. Nach dem Inkrementieren (»6007 INX«) wird durch »6008 CPX # FE« überprüft, ob die Schleife beendet werden kann. Eine eingehendere Beschreibung des Programmablaufs erübrigt sich.

Für Schleifen des Typs a (maximal 256 Durchläufe) ist es aber meist vorteilhaft, eine DEKREMENTIERSCHLEIFE zu verwenden. Wie eine solche Schleife programmiert wird, sehen wir an Listing 13.

Listing 13 unterscheidet sich in der Wirkung nicht von Listing 12, obwohl man dies nicht unbedingt auf den ersten Blick erkennt. Deshalb soll dieses Listing näher besprochen werden. In Zeile 6000 erhält das X-Register den Inhalt $FE. Durch »6002 LDA 01,X« wird damit das letzte Byte der Zeropage, nämlich $FF, zuerst gelesen und nach $70FE geschrieben. Dann wird X dekrementiert. Ist X noch nicht 0, so wird die Schleife erneut durchlaufen.

Der niedrigste X-Wert innerhalb der Schleife ist folglich 1; aufgrund von »6002 LDA 01,X« ist $02 die niedrigste Zeropage-Adresse, die kopiert wird. In Listing 12 ist 0 der niedrigste X-Wert. Die niedrigste Adresse aufgrund von »6002 LDA 02,X« ist also auch $02 (stimmt auffällig). Warum $FF die höchste kopierte Zeropage-Adresse ist, können Sie nun selbst den Listings 12 und 13 entnehmen.

Listing 14 ist eine Dekrementierschleife, die die Kopie der Zeropage wieder von $6F00 nach $02 zurückholt.

Der Vorteil von Dekrementierschleifen beim Typ a ist, daß zum Erkennen der Abbruchbedingung (X=0) kein Vergleichsbefehl erforderlich ist, weil nach dem DEX-Befehl automatisch das Z-Flag gesetzt wird, wenn X Null wird.

Das Entfallen des Vergleichsbefehls »CPX #« bringt eine Ersparnis von 2 Byte Speicherplatz sowie insgesamt 508 Taktzyklen Rechenzeit. Da jedoch bei 6004 eine Seitenüberschreitung (eine Seite entspricht 256 Byte) vorliegt, schrumpft der Zeitgewinn auf 254 Taktzyklen (dies ließe sich aber vermeiden, indem wir die Zeropage nach $6F01 kopieren, womit durch »6004 STA $6F00,X« keine Seitenüberschreitung auftreten würde).

Nun wollen wir noch einen Sonderfall behandeln:

Dekrementierschleifen vom Typ a, bei denen der Ausgangswert für X < 129 ist.

In Listing 15 sehen Sie eine Schleife, die den Bereich $16 - $4A nach $6F00 kopiert, Listing 16 schreibt die Werte von $6F00 zurück nach $16. Selbstverständlich hätten wir das Problem auch so lösen können wie in Listing 13. Wir wollen aber noch eine andere Konstruktion von Dekrementierschleifen kennenlernen, die in diesem Sonderfall möglich ist. Besprechen wir also Listing 15.

Bei 6000 wird ins X-Register die Zahl geladen, die man zu $16 addieren muß, um $4A zu erhalten. Dadurch wird zunächst bei 6002 die Adresse $4A gelesen und nach $6F34 geschrieben. Bei 6007 wird dekrementiert. Neu ist der Verzweigungsbefehl: es wird das N-Flag überprüft. Ist X = $FF, wird das N-Flag gesetzt und »6008 BPL 6002« beendet die Schleife. Der niedrigste Wert von X, der innerhalb der Schleife vorkommt, ist demnach $00.

Der BPL-Befehl funktioniert nur, wenn der Ausgangswert von X <129 ist. Andernfalls wäre nämlich nach dem Dekrementieren X>127 und damit das N-Flag gesetzt. Dies aber hätte zur Folge, daß die Schleife nur 1mal durchlaufen würde.

Zur soeben behandelten Schleifenkonstruktion sind noch zwei Dinge zu sagen; erstens, daß sie nur in diesem Sonderfall (X<129) möglich ist, und zweitens, daß sie nicht effektiver als eine Lösung wie in Listing 13 ist.

Allgemeine Gültigkeit hat aber folgende Regel für Schleifen vom Typ a:

> Bei Schleifen vom Typ a ist Dekrementieren effektiver als Inkrementieren, solange die Durchlaufzahl nicht 255 überschreitet.
> Bei 256 Durchläufen erweist sich Inkrementieren oft als besser.

```
,6000  A0 00     LDY #00
,6002  B9 00 20  LDA 2000,Y
,6005  49 FF     EOR #FF
,6007  99 00 20  STA 2000,Y
,600A  C8        INY
,600B  D0 F5     BNE 6002
,600D  EE 04 60  INC 6004
,6010  EE 09 60  INC 6009
,6013  AD 09 60  LDA 6009
,6016  C9 40     CMP #40
,6018  D0 E8     BNE 6002
```
Listing 24

```
,6000  A0 00     LDY #00
,6002  B9 00 40  LDA 4000,Y
,6005  49 FF     EOR #FF
,6007  99 00 40  STA 4000,Y
,600A  C8        INY
,600B  D0 F5     BNE 6002
,600D  EE 04 60  INC 6004
,6010  EE 09 60  INC 6009
,6013  AD 09 60  LDA 6009
,6016  C9 40     CMP #40
,6018  D0 E8     BNE 6002
```
Listing 25

```
,6000  A9 00     LDA #00
,6002  8D 13 60  STA 6013
,6005  8D 18 60  STA 6018
,6008  A9 20     LDA #20
,600A  8D 14 60  STA 6014
,600D  8D 19 60  STA 6019
,6010  A0 00     LDY #00
,6012  B9 FF FF  LDA FFFF,Y
,6015  49 FF     EOR #FF
,6017  99 FF FF  STA FFFF,Y
,601A  C8        INY
,601B  D0 F5     BNE 6012
,601D  EE 14 60  INC 6014
,6020  EE 19 60  INC 6019
,6023  AD 19 60  LDA 6019
,6026  C9 40     CMP #40
,6028  D0 E8     BNE 6012
```
Listing 26

An Listing 17 sehen wir ein Beispiel für den letzten Satz der Regel. Listing 17 kopiert die letzten 256 Speicherplätze des Stapels ($0100 - $01FF) und den Stapelzeiger nach $6F00 - $7000. Listing 18 schreibt den Stapel wieder zurück.

Die Dekrementierschleife (6000 - 600A) kopiert nun den Bereich $0101 - $01FF, $0100 wird nicht übertragen. Dies geschieht in 600B - 600F. Eine andere Möglichkeit wäre ein zeitraubender CPX #FF-Befehl nach »6008 DEX«.

6011 - 6013 sichert schließlich noch das SP-Register.

Hier ist in der Tat eine Inkrementierschleife besser. Ändern wir Listing 17 also in Listing 17a:

```
         LDX #00
 - LOOP  LDA 0100,X
 -       STA 6F00,X
         INX          ;(!!)
 -       BNE LOOP
 -       TSX
 -       STX 7000
```

Analog ergibt sich Listing 18a:

```
         LDX #00
 - LOOP  LDA 6F00,X
 -       STA 0100,X
         INX          ;(!!)
 -       BNE LOOP
 -       LDX 7000
 -       TXS
```

In den Listings 17a und 18a habe ich diejenigen Befehle, die sich in der symbolischen Darstellung nicht von den Listings 17 und 18 unterscheiden, mit einem »-« markiert.

Typ b: Schleifen mit mehr als 256 Durchläufen

Während Schleifen des Typs a meist so schnell abgearbeitet werden, daß man es gar nicht bemerkt, dauern Typ-b-Schleifen oft eine oder mehrere Sekunden.

Deswegen wollen wir hier versuchen, den Zeitbedarf von Typ-b-Schleifen zu verringern.

Unsere erste Typ-b-Schleife (Listing 19) soll den Bereich von $3FD2 bis $475F invertieren (= EOR #FF-verknüpfen, aus jeder 1 wird eine 0 und umgekehrt). Da hierfür ein 8-Bit-Indexregister nicht ausreicht, benötigen wir einen 16-Bit-Zähler, nämlich $14/$15. Dieser soll immer die Adresse beinhalten, die invertiert wird. In diesen Zähler schreibt die Initialisierung der Schleife den Startwert $3FD2 (siehe $6000 - $6007).

Da es beim 6510 keine indirekte Adressierung für LDA/STA gibt, sondern nur die indirekt-indizierte oder indiziert-indirekte, müssen wir auf eine dieser Adressierungen ausweichen und den Index auf 0 setzen (»6008 LDY #00«).

Bei $600A beginnt die Schleife: der Wert wird eingelesen, mit $FF EOR-verknüpft und zurückgeschrieben. Nun wird der 16-Bit-Zähler $14/$15 erhöht (6010 - 6015). Dann wird

```
,6000  A9 D2     LDA #D2
,6002  8D 11 60  STA 6011
,6005  8D 16 60  STA 6016
,6008  A9 3F     LDA #3F
,600A  8D 12 60  STA 6012
,600D  8D 17 60  STA 6017
,6010  AD 00 00  LDA 0000
,6013  49 FF     EOR #FF
,6015  8D 00 00  STA 0000
,6018  EE 11 60  INC 6011
,601B  EE 16 60  INC 6016
,601E  D0 06     BNE 6026
,6020  EE 12 60  INC 6012
,6023  EE 17 60  INC 6017
,6026  AD 11 60  LDA 6011
,6029  C9 60     CMP #60
,602B  AD 12 60  LDA 6012
,602E  E9 47     SBC #47
,6030  90 DE     BCC 6010
```

Listing 27

```
,6000  A2 00     LDX #00
,6002  8E 11 60  STX 6011
,6005  8E 14 60  STX 6014
,6008  A2 A0     LDX #A0
,600A  8E 12 60  STX 6012
,600D  8E 15 60  STX 6015
,6010  AE 00 00  LDX 0000
,6013  8E 00 00  STX 0000
,6016  EE 11 60  INC 6011
,6019  EE 14 60  INC 6014
,601C  D0 F2     BNE 6010
,601E  EE 12 60  INC 6012
,6021  EE 15 60  INC 6015
,6024  AE 12 60  LDX 6012
,6027  E0 C0     CPX #C0
,6029  D0 E5     BNE 6010
```

Listing 28

```
80   -.BA $6000
90   -.LI 1,3,0
100  -;
110  -; HYPRA-ASS-QUELLTEXT ZU EINER
120  -; SELBSTMODIFIZIERENDEN SCHLEIFE
130  -; (ARBEITET WIE LISTING 5)
140  -;
150  -; 1985 BY FLORIAN MUELLER
160  -;
170  -;
180  -.GL START = $A000
190  -.GL ENDE  = $BFFF
200  -;
210  -            LDX #<(START)
220  -            STX MOD1+1
230  -            STX MOD2+1
240  -            LDX #>(START)
250  -            STX MOD1+2
260  -            STX MOD2+2
270  -MOD1        LDX $FFFF
280  -MOD2        STX $FFFF
290  -            INC MOD1+1
300  -            INC MOD2+1
310  -            BNE MOD1
320  -            INC MOD1+2
330  -            INC MOD2+2
340  -            LDX MOD1+2
350  -            CPX #>(ENDE+1)
360  -            BNE MOD1
```

Listing 29

```
100  -.BA $0801
110  -.OB "LOADER-MAKER 64,P,W"
120  -;
130  -;
140  -; ****************************
150  -; *                          *
160  -; *  L O A D E R - M A K E R *
170  -; *                          *
180  -; ****************************
190  -; *                          *
200  -; *   EIN PROGRAMMGENERATOR  *
210  -; *                          *
220  -; *    VON FLORIAN MUELLER   *
230  -; *                          *
240  -; ****************************
250  -;
260  -;
270  -;
280  -.GL BASIN  = $FFCF
290  -.GL SETPAR = $FFBA
300  -.GL SETNAM = $FFBD
310  -.GL LOAD   = $FFD5
320  -.GL READY  = $A474
330  -.GL NUMOUT = $BDCD
340  -.GL TASTPF = 631 ; TASTATURPUFFER
350  -.GL ANZAHL = 198 ; ENTHAELT ANZAHL
360  -;                  DER ZEICHEN IM
370  -;                  TASTATURPUFFER
380  -.GL KASSPF = 828 ; KASSETTENPUFFER
390  -;
400  -;
410  -.MA PRINT (TEXT)
420  -           LDA #<(TEXT)   ; MAKRO
430  -           LDY #>(TEXT)   ; FUER
440  -           JSR $AB1E      ; TEXTAUSGABE
450  -.RT
460  -;
470  -;
480  -;
490  -;
500  -.WO LINK+1 ; LINKPOINTER
510  -.WO 1985   ; ZEILENNUMMER
520  -.BY $9E    ; TOKEN FUER "SYS"
530  -           .TX "2061"
540  -LINK       .BY 0,0,0      ; ENDMARKIERUNG
550  -;                           DER BASIC-ZEILE
560  -;
570  -SYSTEM     LDX #0         ; FLAG FUER SYSTEM-
580  -           STX $9D        ; MELDUNGEN SETZEN
590  -;
600  -           LDX #$49       ; DEKR.-ZAEHLER
610  -SCHLEIFE1  LDA ABLAGE,X   ; LADEROUTINE
620  -           STA KASSPF,X   ; VON ABLAGE IN
630  -           DEX            ; DEN BEREICH
640  -           BPL SCHLEIFE1  ; KOPIEREN, IN
650  -;                           DEM SIE LAEUFT
660  -           JMP KASSPF     ; & STARTEN
670  -;
680  -;
690  -; ES FOLGT DIE LADEROUTINE, DIE HIER
700  -; AN FALSCHER STELLE ABGELEGT IST UND
710  -; VON DER "SCHLEIFE1" (600-640) IN
720  -; DEN ORIGINALBEREICH GESCHRIEBEN WIRD.
730  -;
740  -ABLAGE     LDA #1         ; FILENUMMER #1
750  -           TAY            ; SEKUNDAERADRESSE #1
760  -GERAETENR  LDX #0         ; GERAETEADRESSE #?
770  -           JSR SETPAR     ; PARAMETER SETZEN
780  -;
790  -LAENGE     LDA #0         ; LAENGE DES FILENAMEN
800  -           LDX #<($35C)   ; ADRESSE DES
810  -           LDY #>($35C)   ; FILENAMEN: $035C
820  -           JSR SETNAM     ; NAMEN SETZEN
830  -;
840  -           LDA #0         ; FLAG FUER "LADEN"
850  -           JSR LOAD
860  -;
870  -FEHLER     BCS LOADERROR  ; LADEFEHLER?
880  -START      JMP 0          ; ZUR STARTADRESSE
890  -LOADERROR  LDX #$1D       ; "LOAD ERROR"
900  -           JMP ($300)     ; AUSGEBEN
910  -;
920  -NAME       .BY 0,0,0,0    ; 16 BYTES
930  -           .BY 0,0,0,0    ; FUER FILENAMEN
940  -           .BY 0,0,0,0    ; RESERVIEREN
950  -           .BY 0,0,0,0
960  -;
970  -BASIC      STX $2D        ; POINTER FUER
980  -           STY $2E        ; PROGRAMMENDE SETZEN
990  -           JSR $E544      ; = PRINT CHR$(147)
1000 -           LDX #3         ; 3 BYTES IN
1010 -           STX ANZAHL     ; TASTATURPUFFER
1020 -;
1030 -SCHLEIFE2  LDA $0383,X    ; AUS DER TABELLE
1040 -           STA TASTPF,X   ; IN ZEILE 1100
1050 -           DEX            ; KOPIEREN
1060 -           BPL SCHLEIFE2
1070 -;
1080 -           JMP READY      ; WARMSTART
1090 -;
1100 -.BY "R",$D5,13 ; "R",SHIFT U,RETURN
1110 -;
1120 -; HIER ENDET DER PROGRAMMTEIL,
1130 -; DER MODIFIZIERT WIRD.
1140 -; ES FOLGT DIE MODIFIKATIONSROUTINE:
1150 -;
1160 -MDFIKATOR JSR $E544       ; = PRINT CHR$(147)
1170 -...PRINT (TEXT1)
1180 -; STARTADRESSE HOLEN
1190 -;
1200 -           JSR $AEFD      ; PRUEFT AUF KOMMA
1210 -           JSR $AD8A      ; HOLT PARAMETER
1220 -           JSR $B7F7      ; NACH $14/$15
1230 -;
1240 -           LDX $14        ; STARTADRESSE
1250 -           LDA $15        ; HOLEN,
1260 -           STX START+1    ; IM PROGRAMM
1270 -           STA START+2    ; ABLEGEN UND
1280 -           JSR NUMOUT     ; UND AUSGEBEN
1290 -;
1300 -;
1310 -; NUN WIRD NOCH DER ZU MODIFIZIERENDE
1320 -; PROGRAMMTEIL IN DEN AUSGANGSZUSTAND
1330 -; GEBRACHT:
1340 -;
1350 -           LDX #15        ; NAMEN MIT NULL-BYTES
1360 -           LDA #0         ; BELEGEN
1370 -SCHLEIFE3  STA NAME,X     ; DURCH EINE
1380 -           DEX            ; DEKREMENTIER-
1390 -           BPL SCHLEIFE3  ; SCHLEIFE
1400 -;
1410 -           STA SYSTEM+1   ; KEINE SYSTEMMELDUNGEN
1420 -;
1430 -           LDA #3         ; SPRUNGWEITE = 3
1440 -           STA FEHLER+1
1450 -;
1460 -           LDA #$A2       ; OPCODE FUER "LDX #"
1470 -           STA GERAETENR
1480 -;
1490 -;
1500 -; AN DIESER STELLE IST DAS "GERUEST"
1510 -; (DER ZU MODIFIZIERENDE TEIL)
1520 -; IM AUSGANGSZUSTAND
1530 -;
1540 -;
1550 -; EINGABE DES FILENAMEN
1560 -; ====================
1570 -;
1580 -...PRINT (TEXT2)
1590 -           LDX #0         ; ZAEHLER AUF 0
1600 -SCHLEIFE4  JSR BASIN
1610 -           CMP #13        ; ENDE DER EINGABE?
1620 -           BEQ WEITER1    ; JA=>WEITER
1630 -           STA NAME,X     ; BYTE ABLEGEN
1640 -           INX
1650 -           CPX #16        ; 16 ZEICHEN MAX.
1660 -           BNE SCHLEIFE4  ; NAECHSTES ZEICHEN
1670 -;
1680 -; WENN DIESE STELLE DURCHLAUFEN WIRD,
1690 -; HAT DAS X-REGISTER DEN WERT 16.
1700 -;
1710 -; BEI "WEITER1" HINGEGEN KANN ES AUFGRUND
1720 -; DES BRANCH-BEFEHLS "BEQ WEITER1"
1730 -; UNTERSCHIEDLICHE WERTE HABEN.
1740 -;
1750 -WEITER1    STX LAENGE+1
1760 -;
1770 -;
1780 -; EINGABE DER GERAETEADRESSE
1790 -; =========================
1800 -;
1810 -...PRINT (TEXT3)
1820 -           JSR BASIN      ; HOLT ZEICHEN
1830 -           SEC            ; VOR SUBTRAKTION
1840 -           SBC #"0"       ; IM AKKU STEHT JETZT
1850 -;                           DIE ZAHL
1860 -;
1870 -           STA GERAETENR+1; ABLEGEN
1880 -           BNE WEITER2    ; GERAET<>0 : WEITER
1890 -; DA ALS GERAETENUMMER 0 EINGEGEBEN
1900 -; WURDE, MUSS DER GESAMTE BEFEHL
1910 -; "LDX #GERAET" IN "LDX $BA"
1920 -; UMGEWAENDELT WERDEN, DAMIT DAS
1930 -; NACHLADEN VON DEM GERAET ERFOLGT,
1940 -; VON DEM DER LADER EINGELESEN WIRD.
1950 -;
1960 -           LDA #$A6       ; OPCODE FUER "LDX ZP"
1970 -           STA GERAETENR
1980 -           LDA #$BA       ; "LDX $BA"
1990 -           STA GERAETENR+1; GENERIEREN
2000 -;
2010 -;
2020 -; MASCHINENPROGRAMM (J/N)?
2030 -; ========================
2040 -;
2050 -WEITER2    ... PRINT(TEXT4)
```

Listing 30





```
2060 -          JSR JANEIN     ; (JA/NEIN)?
2070 -          BEQ WEITER3    ; JA=>WEITER
2080 -          LDA #$6C       ; SPRUNG AUF $036C
2090 -          LDY #$03       ; VERBIEGEN
2100 -          STA START+1    ; BEI $36C STEHT
2110 -          STY START+2    ; EINE ROUTINE,
2120 -;                          DIE DEN "RUN"-
2130 -;                          BEFEHL SIMULIERT
2140 -;
2150 -;
2160 -; SYSTEMMELDUNGEN (J/N)?
2170 -; =====================
2180 -;
2190 -WEITER3    ... PRINT(TEXT5)
2200 -          JSR JANEIN     ; (JA/NEIN)?
2210 -          BNE WEITER4    ; NEIN=>WEITER
2220 -          LDA #$80       ; FLAG FUER
2230 -          STA SYSTEM+1   ; SYSTEMMELDUNGEN
2240 -;
2250 -;
2260 -; LOAD ERROR  AUSGEBEN (J/N)
2270 -; =========================
2280 -;
2290 -;
2300 -WEITER4    ... PRINT(TEXT6)
2310 -          JSR JANEIN     ; (JA/NEIN)?
2320 -          BEQ WEITER5    ; NEIN=>WEITER
2330 -          LDA #0         ; FEHLERMELDUNGEN
2340 -          STA FEHLER+1   ; UNTERDRUECKEN
2350 -;
2360 -;
2370 -; PROGRAMMENDE
2380 -; ============
2390 -;
2400 -;
2410 -WEITER5    ... PRINT(TEXT7)
2420 -;
2430 -; VEKTOR FUER BASIC-ENDE SETZEN
2440 -; ============================
2450 -;
2460 -;
2470 -          LDA #<(MDFIKATOR)
2480 -          STA $2D        ; LOW-BYTE
2490 -          LDA #>(MDFIKATOR)
2500 -          STA $2E        ; HIGH-BYTE
2510 -          JMP READY      ; SPRUNG INS BASIC
2520 -;
2530 -;
10000-;
10010-; ASCII-TABELLEN
10020-; ==============
10030-;
10040-;
10050-TEXT1     .TX "LOADER-MAKER 64"
10060-          .BY 13,13
10070-          .TX "STARTADRESSE : "
10080-          .BY 0
10090-;
10100-TEXT2     .BY 13,13
10110-          .TX "FILENAME : "
10120-          .BY 0
10130-;
10140-TEXT3     .BY 13,13
10150-          .TX "GERAETENR. (1-9;0=UEBERNEHMEN) : "
10160-          .BY 0
10170-;
10180-TEXT4     .BY 13,13
10190-          .TX "MASCHINENPROGRAMM"
10200-          .BY 0
10210-;
10220-TEXT5     .BY 13,13
10230-          .TX "SYSTEMMELDUNGEN"
10240-          .BY 0
10250-;
10260-TEXT6     .BY 13,13
10270-          .TX "LOAD ERROR  AUSGEBEN"
10280-          .BY 0
10290-;
10300-TEXT7     .BY 13,13,18
10310-          .TX "*** LOADER GENERIERT ***"
10320-          .BY 13,13
10330-          .TX "MIT 'SAVE' SPEICHERN,"
10340-          .TX " MIT 'RUN' STARTEN"
10350-          .BY 0
10360-;
10370-TEXT8     .BY 13,13,18
10380-          .TX "*** PROGRAMMENDE ! ***"
10390-          .BY 13,13,0
10400-;
10410-TEXT9     .TX " (J/N)? "
10420-          .BY 0
10430-;
10440-;
20000-;
20010-; UNTERPROGRAMM FUER "J/N?"
20020-; ========================
20030-;
20040-;
20050-JANEIN    ... PRINT(TEXT9)
20060-          JSR BASIN      ; EINGABE HOLEN
20070-          CMP #"←"
20080-          BNE JANEIN1
20090-          PLA            ; SIEHE STAPEL-
20100-          PLA            ; MANIPULATION
20110-...PRINT(TEXT8)
20120-          JMP READY      ; SPRUNG INS BASIC
20130-JANEIN1   CMP #"J"       ; VERGLEICH MIT "J"
20140-          RTS            ; RUECKKEHR VOM
20150-;                          UNTERPROGRAMM
20160-.EN
```

Listing 30 (Schluß)

überprüft, ob die nächste Adresse schon mit der ersten Adresse nach der Endadresse ($475F), also $4760, übereinstimmt (siehe $6016 - $601D). Dieser 16-Bit-Vergleich wurde bereits im SMON vorgestellt. Bei $601E wird schließlich die Schleife beendet, falls die Abbruchbedingung (C=1) erfüllt ist.

Listing 20 ist eine Dekrementierschleife, die sich in der Wirkung nicht von Listing 19 unterscheidet. Da das Dekrementieren einer 16-Bit-Adresse beim 6510 langsamer und speicherplatzaufwendiger ist als das Inkrementieren, ist Listing 20 weniger effektiv als Listing 19.

Grundsätzlich können Sie an den Listings 19 und 20 sehen, wie man eine Typ-b-Schleife programmiert. Diese arbeitet jedoch nicht besonders schnell. Der Grund ist, daß der Bereich von $3FD2 - $475F nicht restlos in ganze Seiten (256-Byte-Blöcke)aufgeteilt werden kann. Daher sollte man sich immer überlegen, ob die Schleifendurchlaufzahl nicht auf ganze 256-Byte-Blöcke »aufgerundet« werden kann. In unserem Fall würde dies heißen, daß mit einer schnelleren Schleife der exakt 8 x 256 Byte lange Bereich $3FD2 - $47D1 invertiert wird, anstelle des »ungeraden« Bereichs $3FD2 - $475F. An einfacheren Zahlen wollen wir nun eine solche Schleife für ganze Seiten programmieren. Der 32 x 256 Byte umfassende Bereich von $2000 bis $3FFF (einschließlich) soll invertiert werden. Mit einer solcher Routine könnte das gerade sichtbare Bild bei Hi-Eddi invertiert werden.

Die einfachste Form finden Sie in Listing 21. Zuerst wird die Anfangsadresse in $14/$15 abgelegt. Ins Y-Register kommt der Wert 0. Dann wird der Wert invertiert und das Y-Register, der Low-Zähler, erhöht. Ist der Wert noch nicht 0, wird die Schleife neu durchlaufen. Andernfalls wurde gerade eine Seite abgearbeitet. Der High-Zähler ($15) wird erhöht. Ist der Inhalt des High-Zählers = $40, wird die Schleife abgebrochen. Zu bemerken ist, daß während der Schleife die Adresse $14 unverändert 0 bleibt. Die Adresse, die invertiert wird, ergibt sich folgendermaßen:

(Y+Inhalt von $14)+256*(Inhalt von $15)

Da wir auf die Adresse über das Prozessor-Register Y Einfluß nehmen können und die Adresse $14 nicht verändert werden muß, ist die Verarbeitungsgeschwindigkeit gegenüber der »Normalform« (Listing 20) gestiegen. Das High-Byte müssen wir aber weiterhin in $15 belassen. Neu führen wir den High-Zähler X ein. Im X-Register merken wir uns, wieviele Seiten invertiert werden. Diesen Wert verwenden wir als Dekrementierzähler. In unserem Fall werden $20 Seiten invertiert. Weil $20 zufälligerweise auch das High-Byte der Anfangsadresse ($2000) ist, wird dieser Wert in Listing 22 nur einmal (6005) in den Akku geladen und dann bei 6009 ins X-Register übertragen.

Beachten Sie bitte, daß in Listing 22 die Befehle »6004 TAY« und »6009 TAX« nur bei den Werten dieses Beispiels verwendet werden können. In der Regel sind eigene »LDX #«- oder »LDY #«-Befehle erforderlich. Wenn wir zum Bei-

spiel den Bereich $3FD2 - $47D1 invertieren wollen, muß die Initialisierung so aussehen:

```
LDA #D2   Low-Byte der ersten Adresse
STA 14
LDY #00   Index-Register
LDA #3F   High-Byte der ersten Adresse
STA 15
LDX #08   High-Zähler
```

... Schleife wie ab 600C in Listing 22

Damit hätten wir eine Schleife, die den Bereich #3FD2 - $475F (siehe Listings 19 und 20) invertiert und wesentlich schneller als die Listings 19 und 20 arbeitet. Da wir aber »aufgerundet« haben, wird zusätzlich der Bereich $4760 - $47D1 invertiert, obwohl wir das gar nicht wollen. Es gibt nun mehrere Möglichkeiten, dies zu verhindern:

1. Wir verwenden die Schleife aus Listing 19, müssen aber eine deutlich höhere Arbeitsdauer hinnehmen.

2. Wir verwenden die Schleife aus Listing 22 mit obiger Initialisierung. Dann invertiert eine Typ-a-Schleife den Restbereich $4760 - $47D1 ein weiteres Mal. Damit wären - eine Besonderheit der EOR #FF-Verknüpfung - im Restbereich die alten Inhalte wiederhergestellt. Diese Lösung eignet sich aber (fast) nur bei dieser logischen Verknüpfung und hilft bei den meisten anderen Typ-b-Schleifen nicht weiter.

3. Dies dürfte wohl die beste Lösung sein: Wir schreiben eine »gemischte« Schleife, die aus einer Typ-a-Schleife und einer Typ-b-Schleife besteht. Dieses Verfahren ist immer (!) möglich und wird von der BLTUC-Routine ($A3BF) des Basic-Interpreters angewandt. Diese Verschiebe-Routine zerlegt den Bereich, der verschoben werden soll, in einen Bereich der aus 256-Byte-Blöcken besteht und in einen Restbereich. Beide Bereiche werden dann getrennt verschoben.

Folgendermaßen sieht die optimale Invertierroutine für den Bereich $3FD2 - $475F aus:

a) Der exakt 7 Seiten umfassende Bereich 3FD2 - $46D1 wird mit einer Typ-b-Schleife wie in Listing 22 komplementiert.

b) Der Restbereich $46D2 - $475F wird mit einer Typ-a-Schleife wie in Listing 13 komplementiert.

Wir haben nun viele verschiedene Schleifenkonstruktionen in Theorie und Praxis behandelt. Was uns noch fehlt, sind Formeln, nach denen Sie die einzelnen Parameter (zum Beispiel den Startwert für X in einer Dekrementier-Schleife vom Typ a) errechnen können. Als Zusammenfassung finden Sie in Form von Listing 23 ein Hypra-Ass-Assemblerlisting zu mehreren Schleifenkonstruktionen. An den Quelltext-Ausdrücken können Sie sehen, wie einzelne Parameter errechnet werden können.

Merke: Sofern es der Programmablauf zuläßt, sollten Sie Inkrementierschleifen verwenden.

Bei Verschiebeschleifen ist aber oft eine Dekrementierschleife erforderlich.

Noch etwas zum Schleifen-Inhalt: Wenn mehrere Schleifen einen gleichen Innenteil haben (zum Beispiel einen Invertierbefehl), definieren Sie diesen unbedingt als Makro und nicht als Unterprogramm! JSRs sollten Sie nur beim Aufruf von ROM-Routinen verwenden.

Damit wäre das Thema »Schleifen« erst einmal abgeschlossen. Im nächsten Abschnitt (über Selbstmodifikation) werden wir uns aber wieder mit Schleifen auseinandersetzen.

```
programm : loader-maker      0801 0a38
-------------------------------------

0801 : 0b 08 c1 07 9e 32 30 36   0a
0809 : 31 00 00 00 a2 00 86 9d   ba
0811 : a2 49 bd 1f 08 9d 3c 03   0f
0819 : ca 10 f7 4c 3c 03 a9 01   f7
0821 : a8 a2 00 20 ba ff a9 00   71
0829 : a2 5c a0 03 20 bd ff a9   c5
0831 : 00 20 d5 ff b0 03 4c 00   0b
0839 : 00 a2 1d 6c 00 03 00 00   77
0841 : 00 00 00 00 00 00 00 00   42
0849 : 00 00 00 00 00 00 86 2d   be
0851 : 84 2e 20 44 e5 a2 03 86   09
0859 : c6 bd 83 03 9d 77 02 ca   72
0861 : 10 f7 4c 74 a4 52 d5 0d   5d
0869 : 20 44 e5 a9 21 a0 09 20   d5
0871 : 1e ab 20 fd ae 20 8a ad   9e
0879 : 20 f7 b7 a6 14 a5 15 8e   37
0881 : 38 08 8d 39 08 20 cd bd   7c
0889 : a2 0f a9 00 9d 3f 08 ca   a7
0891 : 10 fa 8d 0e 08 a9 03 8d   38
0899 : 36 08 a9 a2 8d 22 08 a9   ef
08a1 : 42 a0 09 20 1e ab a2 00   44
08a9 : 20 cf ff c9 0d f0 08 9d   9e
08b1 : 3f 08 e8 e0 10 d0 f1 8e   b7
08b9 : 28 08 a9 50 a0 09 20 1e   69
08c1 : ab 20 cf ff 38 e9 30 8d   1f
08c9 : 23 08 d0 0a a9 a6 8d 22   b0
08d1 : 08 a9 ba 8d 23 08 a9 74   10
08d9 : a0 09 20 1e ab 20 1b 0a   06
08e1 : f0 0a a9 6c a0 03 8d 38   97
08e9 : 08 8c 39 08 a9 88 a0 09   fa
08f1 : 20 1e ab 20 1b 0a d0 05   5f
08f9 : a9 80 8d 0e 08 a9 9a a0   81
0901 : 09 20 1e ab 20 1b 0a f0   fc
0909 : 05 a9 00 8d 36 08 a9 b1   42
0911 : a0 09 20 1e ab a9 69 85   ba
0919 : 2d a9 08 85 2e 4c 74 a4   2e
0921 : 4c 4f 41 44 45 52 2d 4d   24
0929 : 41 4b 45 52 20 36 34 0d   4a
0931 : 0d 53 54 41 52 54 41 44   7a
0939 : 52 45 53 53 45 20 3a 20   ec
0941 : 00 0d 0d 46 49 4c 45 4e   7d
0949 : 41 4d 45 20 3a 20 00 0d   45
0951 : 0d 47 45 52 41 45 54 45   b8
0959 : 4e 52 2e 20 28 31 2d 39   93
0961 : 3b 30 3d 55 45 42 45 52   ce
0969 : 4e 45 48 4d 45 4e 29 20   c1
0971 : 3a 20 00 0d 0d 4d 41 53   44
0979 : 43 48 49 4e 45 4e 50 52   a9
0981 : 4f 47 52 41 4d 4d 00 0d   8a
0989 : 0d 53 59 53 54 45 4d 4d   40
0991 : 45 4c 44 55 4e 47 45 4e   89
0999 : 00 0d 0d 4c 4f 41 44 20   3d
09a1 : 45 52 52 4f 52 20 20 41   b7
09a9 : 55 53 47 45 42 45 4e 00   aa
09b1 : 0d 0d 12 2a 2a 2a 20 4c   1c
09b9 : 4f 41 44 45 52 20 47 45   30
09c1 : 4e 45 52 49 45 52 54 20   e8
09c9 : 2a 2a 2a 0d 0d 4d 49 54   3e
09d1 : 20 27 53 41 56 45 27 20   ee
09d9 : 53 50 45 49 43 48 45 52   ff
09e1 : 4e 2c 20 4d 49 54 20 27   fd
09e9 : 52 55 4e 27 20 53 54 41   cf
09f1 : 52 54 45 4e 00 0d 0d 12   49
09f9 : 2a 2a 2a 20 50 52 4f 47   2a
0a01 : 52 41 4d 4d 45 4e 44 45   53
0a09 : 20 21 20 2a 2a 2a 0d 0d   49
0a11 : 00 20 28 4a 2f 4e 29 3f   fd
0a19 : 20 00 a9 12 a0 0a 20 1e   fd
0a21 : ab 20 cf ff c9 5f d0 0c   c3
0a29 : 68 68 a9 f6 a0 09 20 1e   1e
0a31 : ab 4c 74 a4 c9 4a 60 5c   dd
```

Listing 31

10. Selbstmodifikation

Bevor wir uns mit dieser Programmiertechnik beschäftigen, die zwar nicht strukturiert, aber sehr trickreich ist, soll der Begriff geklärt werden.

Unter Modifikation versteht man »eine Änderung, Anpassung«. Wenn Sie bei einem Spiel einen der vielen POKE-Befehle, die im 64'er schon vorgestellt wurden, eingeben, so wird dadurch das Spiel »modifiziert«. Die Änderung ist zum Beispiel eine Erhöhung der Spielfigurenanzahl.

Selbstmodifikation bedeutet, daß ein Programm sich selbst programmgesteuert verändert. Dies wäre der Fall, wenn im Spielprogramm eine Passage stünde, die den POKE ausführt.

Wenn Sie sich für die Selbstmodifikation von Basic-Programmen interessieren, finden Sie in der Zeitschrift »Happy-Computer« (Ausgabe 8/85) unter der Überschrift »Lernen Sie Ihren Commodore 64 kennen« alles, was Sie wis-

sen müssen. Auf simulierten Direktmodus wurde im 64'er schon mehrfach eingegangen, unter anderem in der »Memory Map mit Wandervorschlägen«.

Wir werden uns an dieser Stelle ausschließlich mit der Selbstmodifikation von Maschinenprogrammen befassen. Als erstes Beispiel nehmen wir Listing 24.

Es handelt sich um eine selbstmodifizierende Schleife, die den Bereich $2000 - $3FFF komplementiert.

TRACEn Sie doch einmal Listing 24 mit dem SMON und vergleichen Sie die disassemblierten Befehle mit den ursprünglichen Werten, die Sie in Listing 24 finden. Sie werden erkennen, daß die Befehle »6002 LDA 2000,Y« und »6007 STA 2000,Y« aufgrund der INC-Befehle immer auf andere Adressen zugreifen. Besagte INC-Befehle erhöhen jeweils das High-Byte des Operanden. Ist dieses schon $40, so wird die Schleife beendet. In Listing 25 sehen Sie, wie unsere Schleife aus Listing 24 aussieht, wenn sie fertig durchlaufen wurde. Ein weiterer Start bewirkt, daß das Programm sich früher oder später selbst invertiert und darum abstürzt.

Was nämlich unserem Listing 24 fehlt, damit es mehr als einmal arbeitet, ist eine Initialisierung, die jedesmal den Ausgangswert ($2000) in die LDA/STA-Befehle einsetzt. In Listing 26 sehen Sie eine solche Initialisierung (6000 - 600F). Die Adresse $FFFF (bei 6012 und 6017) ist ein Dummy-Wert, das heißt er dient nur zum vorläufigen Ausfüllen von Adressen und hat keine programmtechnische Bedeutung. Der Dummy-Wert wird ohnehin von der Initialisierung überschrieben; wir hätten also statt $FFFF auch $040C oder andere verwenden können. Wichtig ist nur, daß »LDA Dummy,Y« 3 Byte belegt.

Ein besonderer Vorteil der Selbstmodifikation ist es, daß selbstmodifizierende Schleifen keine Zähler in der Zeropage benötigen, weil der Zähler praktisch im Programm selbst liegt. In puncto Geschwindigkeit sind selbstmodifizierende Schleifen den herkömmlichen aber oft unterlegen.

Ein weiterer Vorteil von ihnen ist aber, daß man außer mit weniger Zeropage-Speicherplätzen auch mit weniger Registern auskommen kann (sofern man hier Einsparungen vornehmen will). Listing 27 beispielsweise invertiert den Bereich $3FD2 - $475F. X- und Y-Register sowie die Zeropage bleiben unverändert, lediglich der Akkumulator fungiert als Arbeitsregister.

Listing 28 kopiert den Basic-Interpreter ($A000 - $BFFF) ins RAM an gleicher Adresse, wobei nur das X-Register verwendet wird (!).

Nun wollen wir sehen, wie man bei der Entwicklung selbstmodifizierender Programme unter Zuhilfenahme eines guten Assemblers (Hypra-Ass) vorgehen muß.

Zunächst einmal müssen diejenigen Stellen, an denen Modifikationen vorgenommen werden, mit Label definiert werden. Von diesen Label aus können die Stellen im Speicher die geändert werden sollen, leicht berechnet werden.

```
Befehlscode   = LABEL + 0 = LABEL
Low-Operand   =             LABEL + 1
High-Operand  =             LABEL + 2
```

Bei 2-Byte-Befehlen wird der Parameter wie der Low-Operand eines 3-Byte-Befehls errechnet.

Als Beispiel finden Sie in Form von Listing 29 einen Quelltext (Assembler: Hypra-Ass) für Listing 28. Während in Listing 28 der Ausgangswert bei 6010 »LDX 0000« und bei 6013 »STX 0000« ist, wurde im Quelltext $FFFF verwendet (270, 280), um den Assembler zu zwingen, den Dummy-Wert als 16-Bit-Adresse abzulegen (und nicht als Zeropage-Adresse, wodurch der Befehl nur 2 statt 3 Byte belegen würde).

Die Stellen, die modifiziert werden, wurden mit »MOD1« und »MOD2« definiert. MOD1 ist zugleich der Schleifenbeginn.

Nachdem Sie jetzt den Eingang gefunden haben, möchte ich einige Anregungen liefern, wie Sie die Vorteile der Selbstmodifikation nutzen können. Wir werden hier die Anwendung nach den verschiedenen Adressierungsarten unterteilen.

a) Anwendung auf absolute Adressierung

Bei der Stapelmanipulation haben wir schon ein Verfahren kennengelernt, den Befehl JSR (indirekt), der im normalen 6510-Befehlssatz nicht existiert, zu simulieren.

Folgendermaßen kann über Selbstmodifikation ein Unterprogramm ab ADRESSE aufgerufen werden.

```
               LDA #<ADRESSE
               STA SPRUNGBEFEHL+1 ; Low-Operand
               LDA #>ADRESSE
               STA SPRUNGBEFEHL+2 ;
               High-Operand
SPRUNGBEFEHL
               JSR $FFFF ; $FFFF=Dummy
```

Genauso kann man mit dem JMP-Befehl verfahren. Sogar bei den Schieber-, Dekrementier- und Inkrementierbefehlen, die im Gegensatz zu JMP die indirekte Adressierung nicht haben, ist auf diese Weise eine Simulation der indirekten Adressierung möglich.

Wird eine Sprungtabelle per Selbstmodifikation verarbeitet, müssen die Sprungadressen in der Tabelle nicht (!) dekrementiert werden.

b) Anwendung auf Immediate-Befehle

Oft müssen Werte, die berechnet werden, auf dem Stapel oder im Speicher abgelegt und dann, wenn sie gebraucht werden, wieder aufgenommen werden.

Ein Beispiel hierfür ist der »Basic-Start-Generator« (64'er, 7/85, Seite 74). Bei Erwähnung dieses Programms taucht natürlich die Frage auf, ob es sich hier noch um ein selbstmodifizierendes Programm handelt oder ob der »Basic-Start-Generator« nicht eher zu den Programmgeneratoren zählt. Diese Frage ist voll berechtigt. Deshalb wollen wir darauf kurz eingehen.

Der »Basic-Start-Generator« ist eindeutig den Programmgeneratoren zuzuordnen, da der generierte Programmteil nie angesprungen wird und somit ein eigenständiges Programm darstellt. Das Programm modifiziert also nicht sich selbst, sondern vielmehr ein zweites Programm, welches dann vom Benutzer gespeichert werden kann.

Die Programmierung ist aber bei Programmgeneratoren nicht anders als bei selbstmodifizierenden Programmen. Auf den Unterschied Programmgeneration/Selbstmodifikation werden wir an späterer Stelle näher eingehen.

Zunächst wollen wir aber ein praktisches Beispiel für die Anwendung der Modifikation von Immediate-Befehlen behandeln. Oft steht man vor dem Problem, ein Register zu sichern und später wieder zu holen. Im Falle des Akkumulators sieht das so aus:

```
      PHA         ; Akku sichern
      .....       ; weiteres Programm
      PLA         ; Akku wieder holen
```

Beim X-Register wird's schon ungünstiger:

```
      TXA         ; X-Register in Akku
      PHA         ; Akku sichern
      ......      ; weiteres Programm
      PLA         ; Akku wieder holen
      TAX         ; Akku ins X-Register
```

Hier wird also zusätzlich der Akku beeinflußt. Wenn dies vermieden werden muß, wird folgender Weg gewählt:

```
      STX $02     ; $02 = Zwischenspeicher
      ....        ; weiteres Programm
      LDX $02     ; X wieder holen
```

Für die Sicherung des X-Registers gibt es aber noch eine weitere Lösung, die den X-Wert im Programm ablegt und

dadurch nicht den Stapel oder einen Zwischenspeicher außerhalb des Programms benötigt.

```
        STX GETX+1    ; X direkt in Immediate-Befehl
                        schreiben
        ....          ; weiteres Programm
GETX    LDX #$00      ; $00 = Dummy-Wert
```

Obiges Beispiel kann sehr leicht auf Akkumulator oder Y-Register umgeschrieben werden.

Folgendermaßen kann das X-Register mit dem Akkumulator verglichen werden:

```
        STX VGL+1     ; in Vergleichsbefehl ablegen
        (........     ; evtl. weitere Programme)
VGL     CMP #$00      ; $00 = Dummy
```

Als letztes Beispiel für die Anwendung auf Immediate-Befehle soll das Y-Register zum Akkumulator addiert werden:

```
        STY ADD+1     ; in Arithmetikbefehl ablegen
        (.........    ; evtl. weiteres Programm)
        CLC           ; Carry vor Addition
ADD     ADC #$FF      ; $FF = Dummy
```

Die Anwendungsmöglichkeiten sind hier unbegrenzt.

c) Anwendung auf komplette Befehle

Bisher haben wir nur die Parameter einzelner Befehle modifiziert. Es ist selbstverständlich auch möglich, die Befehlscodes oder die kompletten Befehle zu modifizieren.

Wenn nur der Befehlscode geändert wird (zum Beispiel ein ORA #- in einen EOR #-Befehl) bleiben die Parameter erhalten. Es könnte ferner ein impliziter Befehl (SEI,CLI,CLD, DEX,INX,....) geändert werden, um beispielsweise zwischen In- und Dekrementieren umzuschalten. Außerdem könnte bei einem BRANCH-Befehl die Sprungbedingung (CS,CC,VS, VC,NE,EQ) geändert werden. Aus BCS könnte also leicht BCC werden.

Weil man hier die Opcodes der Befehle kennen muß, empfehle ich das erste 64'er Extra (Ausgabe 9/85) oder die Tabelle am Ende dieser Ausgabe.

Nun lösen wir noch das häufig auftretende Problem, wie die Ausführung eines Unterprogramms verhindert wird. Dazu werden wir drei Lösungen (I – III) entwickeln.

I. Die Adresse FLAG wird auf 0 gesetzt, wenn das Unterprogramm ausgeführt werden soll; auf einen anderen Wert, wenn es nicht ausgeführt werden soll.

```
        LDA #0        ; Flag für Ausführung
        STA FLAG      ; Flag setzen
        (........     ; evtl. weiteres Programm)
        LDA FLAG      ; Flag testen
        BNE NEIN      ; Flag < > 0, also nicht ausführen
        JSR UNTER-
        PROGRAMM      ; Aufruf
NEIN    ........      weiteres Programm
```

Das Flag könnte auch am Beginn des Unterprogramms abgefragt und dann (wenn FLAG < > 0) das Unterprogramm verlassen werden.

II. Als ersten Befehl des Unterprogramms verwenden wir NOP:

```
UP      NOP           ; Beginn des Unterprogramms
        ......        ; Fortsetzung des Unterprogramms
```

So wird die Ausführung des Unterprogramms gestattet:

```
        LDA #$EA      ; Opcode für NOP
        STA UP        ; an Anfang des Unterprogramms
                        schreiben
```

Und so wird sie verhindert:

```
        LDA #$60      ; Opcode für RTS
        STA UP        ; an Anfang des Unterprogramms
                        schreiben
```

Wer noch einen NOP-Befehl und damit 1 Byte sparen möchte, kann den NOP-Befehl entfallen lassen. Dann muß auch der Opcode $EA beim Erlauben des Unterprogramms in den Opcode des ersten Byte im Unterprogramm geändert werden. Weil dies ziemlich mühselig ist, ziehe ich die ursprüngliche Lösung II trotz des um 1 Byte erhöhten Speicherbedarfs vor.

III. Das beste Verfahren. Wir schalten den JSR-Befehl aus, indem wir ihn in einen BIT-Befehl abändern.

```
AUFRUF    JSR Unterprogramm
JSR ausschalten:
          LDA #$2C     ; Opcode für BIT
          STA AUFRUF
JSR wieder erlauben:
          LDA #$20     ; Opcode für JSR
          STA AUFRUF
```

Der JSR-Opcode kann auch mit $0C überschrieben werden. $0C ist ein illegaler Opcode für ein 3-Byte-NOP und arbeitet mit allen mir bekannten Versionen des C 64. Ob er ebenfalls auf dem C 128 läuft, konnte ich noch nicht prüfen.

Im übrigen können mit dem soeben beschriebenen Verfahren auch andere Befehle ausgeschaltet werden, zum Beispiel JMP, LDA, STA und so weiter. Wenn aber der JSR-Opcode mit $2C (BIT) überschrieben wird, ist darauf zu achten, daß bei der Ausführung des BIT-Befehls die Prozessorflags gesetzt werden.

Sicherlich gibt es noch mehr Problemlösungen als I – III, aber III dürfte wohl kaum zu übertreffen sein.

d) Anwendung auf mehrere Befehle

Selbstverständlich können ganze Befehlsfolgen, also größere Programmteile gegeneinander ausgetauscht werden. Zu beachten ist nur, daß die Routinen, die gegeneinander ausgetauscht werden, auch in dem Bereich, in den sie vom Programm aus geschrieben werden, lauffähig sind. Dies ist vor allem dann gegeben, wenn nur die relative Adressierung verwendet wird und dadurch die Routine im Speicher frei verschoben werden kann.

e) Anwendung auf Tabellen

Dieser Anwendungsfall würde auch zum Abschnitt über »Tabellen« passen.

Wir bleiben hier bei der Theorie, denn die Umsetzung in ein Programm ist nicht mehr schwer. Vielmehr soll Ihre Kreativität nicht durch Unmengen von Beispielen gehemmt werden.

Zunächst wollen wir uns ein wenig mit dem SMON befassen. Wenn Sie den Disk-Monitor einschalten, kopiert das Programm einen Floppy-Befehl («U1 ..«) vom Ende des SMON in einen Bereich zwischen $02A0 und $02FF. Dieser Lesebefehl wird nach Bedarf modifiziert, zum Beispiel wird beim Schreiben der »U1«- in einen »U2«-Befehl umgewandelt oder die Angabe des einzulesenden Blocks wird geändert. Dies wäre ein typisches Anwendungsbeispiel für Selbstmodifikation, wenn der Lesebefehl nicht erst in einen Bereich außerhalb des Programms kopiert würde (worin ich keinen Sinn sehe), sondern am Ende des SMON (etwa bei $CFF0) bliebe und dort modifiziert würde.

Im Hi-Eddi liegt eine Tabelle, die die High-Byte der Bit-Map-Anfangsadressen beinhaltet. Diese Tabelle wird von Hi-Eddi bei jedem Bildwechsel umgerechnet.

Nach den vorausgegangenen zwei Beispielen an Spitzenprogrammen aus dem 64'er möchte ich noch andere Anwendungsbeispiele nennen.

Besonders flexible Programme erlauben Eingriffe des Anwenders in die Befehls- oder Text-Tabellen. So können Bildschirmmasken editiert oder Eingabemasken erstellt werden.

Ein solches Programm braucht sich nach den Modifikationen nur selbst abzuspeichern. Weil hier unter Umständen ein erheblicher Teil des Programmschutzes verlorengeht, werden dann lediglich die Tabellen gespeichert.

Ein Adventure-Generator modifiziert in der Regel auch nur die Tabellen eines fertigen Adventureprogramms, das eigentliche Programm bleibt unverändert. In diesen Tabellen sind die einzelnen Spielsituationen enthalten.

Bei diesen (theoretischen) Fällen wollen wir es belassen. Letztendlich muß ja der Programmierer entscheiden, inwieweit er die Selbstmodifikation auf Tabellen anwenden kann.

f) Das Beispielprogramm »Loader-Maker 64«

Wie aus dem Namen des Beispielprogramms schon zu entnehmen ist, handelt es sich um einen Programmgenerator. Da – wie gesagt – die Programmierung wie bei selbstmodifizierenden Programmen ist, habe ich bewußt einen Programmgenerator als Beispiel gewählt.

Als Listing 31 finden Sie ein MSE-Listing, falls Sie »Loader-Maker 64« bequem abtippen wollen und an der Anwendung des Programms interessiert sind. Deshalb zunächst eine Kurzbeschreibung für Anwender.

»Loader-Maker« ermöglicht es Ihnen, zu einem Programm ein (Maschinensprache-) Ladeprogramm zu generieren, welches normal geladen und mit »RUN« gestartet wird, worauf es das nachzuladende Programm nachlädt und startet.

Nach dem Laden von »Loader-Maker« wird dieses Programm durch SYS 2154,START gestartet. START ist eine Variable und wird durch die Startadresse des nachzuladenden Programms ausgedrückt. Soll ein Basic-Programm nachgeladen werden, hat diese Adresse keine Bedeutung (einfach SYS2154,0 eingeben). Bei einem Maschinenprogramm handelt es sich hier um die Adresse, mit der das Programm über »SYS« gestartet wird (49152 beim SMON $C000).

Das Programm meldet sich mit »Loader-Maker 64« und gibt die Startadresse aus. Dazu können Sie den Filenamen eingeben.

Bei allen weiteren Eingaben (Gerätenummer, von der geladen werden soll; Maschinenprogramm j/n; Systemmeldungen wie »SEARCHING FOR« ausgeben j/n; LOAD ERROR bei Ladefehler ausgeben j/n) können Sie das Programm durch Eingabe des Linkspfeils abbrechen. Sind alle Eingaben gemacht worden, kommt die Meldung »LOADER GENERIERT« und der Lader kann mit »SAVE« gespeichert werden.

Wenn das nachzuladende Programm von der Adresse geladen werden soll, von der auch das Ladeprogramm selbst eingelesen wurde, ist als Gerätenummer nur 0 einzugeben.

Befassen wir uns nun mit dem Programm, dessen Quelltext Sie als Listing 30 finden.

Die Zeilen bis 990 stellen das Ladeprogramm in unmodifizierter Form dar und enthalten viele Dummywerte, wie zum Beispiel die (unsinnige) Startadresse 0 in Zeile 820.

Mit 1000 beginnt die Modifikationsroutine. Nach 1120 wurde die Startadresse eingelesen, die ja per SYS übergeben wurde, und wird wieder mit dem Titel ausgegeben. 1100/1110 schreiben die Startadresse hinter den JMP-Befehl in Zeile 820.

1150 – 1350 bringen das (noch unmodifizierte) Gerüst in den Ausgangszustand, der dann nach Bedarf geändert wird.

1400 – 1550 holen den Filenamen, legen ihn bei NAME (850) ab, berechnen gleich die Länge des Filenamens und legen diese bei LAENGE (750) ab.

1600 – 1720 holen die Geräteadresse. Da diese im ASCII-Format vorliegt, muß der ASCII-Code von 0 abgezogen werden (1640/1650). Wurde 0 eingegeben, wird der LDX #DEVICE-Befehl (730) in »LDX $BA« geändert. Die Adresse $BA enthält jeweils die Adresse, von der das letzte Programm geladen wurde.

1750 – 1850 fragen, ob das nachzuladende Programm mit der per SYS übermittelten Startadresse gestartet wird (Eingabe »j«). Wurde »n« eingegeben, muß das Programm über den Basic-Befehl RUN eingegeben werden. Auf eine entsprechende Routine (870 – 980) wird die Startadresse gestellt (1810 – 1840).

1900 – 1970 ermöglichen die Einstellung, ob »SEARCHING..«, »LOADING« etc. ausgegeben werden sollen.

Soll im Falle eines Ladefehlers das Programm nicht gestartet und stattdessen »LOAD ERROR..« ausgegeben werden, wird dies bei 2000 – 2090 festgelegt. Wird die Fehlerausgabe unterdrückt, muß der BCS-Befehl (810) unschädlich gemacht werden. Dies geschieht einfach dadurch, daß die Sprungweite auf 0 gesetzt wird (2070/2080).

Am Programmende wird noch eine Meldung ausgegeben (2140 – 2160) und der Vektor für das Ende des Basic-Programms neu gesetzt, damit das generierte Ladeprogramm mit »SAVE« gespeichert werden kann.

10000 – 10310 enthalten nur die Text-Tabellen.

Von 15000 bis zur letzten Zeile (15170) steht ein Unterprogramm, daß bei jeder J/N-Entscheidung über »JSR J,N« aufgerufen wird.

Es gibt den Text »(J/N)?« aus (15030 – 15050) und holt eine Eingabe. Ist diese »J«, so ist nach dem Verlassen des Unterprogramms (1517) das Zero-Flag gesetzt (andernfalls nicht).

Wurde der Linkspfeil eingegeben, wird das Programm abgebrochen und eine entsprechende Meldung ausgegeben (15100 – 15150).

Wie wir nun gesehen haben, handelt es sich bei »Loader-Maker« um einen Programmgenerator. Mit zwei kleinen Änderungen wird er jedoch zum selbstmodifizierenden Ladeprogramm. Wir müssen nur die beiden »JMP READY.«-Befehle (2240/15150) in »JMP SYSTEM« umwandeln, wodurch am Programmende der generierte Lader angesprungen würde. Schon hätten wir ein selbstmodifizierendes Ladeprogramm.

Um Ihnen noch die Anwendung des Loader-Maker zu erleichtern, hier zwei Eingabebeispiele:

```
Startadresse...................49152
Filename.......................SMON $C000
Geräteadresse..................0
Maschinenprogramm .......j
Systemmeldungen...........j
LOAD ERROR ausgeben..j

Startadresse...................0 (bedeutungslos)
Filename.......................HI-EDDI
Geräteadresse..................8
Maschinenprogramm .......n
Systemmeldungen...........n
LOAD ERROR ausgeben..j
```

g) Verbesserungen an »Tabellen-Beispiel«

Zum Abschluß des Themas »Selbstmodifikation« wollen wir noch kleine Verbesserungen am Programm »Tabellen-Beispiel« erwähnen. Ich werde hier eher Anregungen geben als fertige Änderungsvorschläge.

Zunächst soll die Adresse XSAVE (zum Sichern des X-Registers in Schleifen) überflüssig werden. So könnte es nun gesichert werden:

```
XSV   STX  GETX
.......
GETX  LDX  #$00      0=Dummy; hier wird X wieder
                          aufgenommen.
```

Auch die Sprungtabelle läßt sich – viel einfacher, finde ich – anders handhaben:

```
      LDA  J?LO,X     JMLO oder JELO
      STA  SPRO+1
      LDA  J?HI,X     JMHI oder JEHI
      STA  SPRG+2
SPRG  JMP  0000
```

In den Tabellen JMLO/JMHI und JELO/JEHI (Low- und High-Bytes der Sprungadressen) dürfen die Adressen aber nicht dekrementiert werden.

Wird ein JSR (IND)-Befehl simuliert, muß nach wie vor die Rücksprungadresse auf den Stapel gelegt werden. Dies würde entfallen, wenn die Rücksprungadresse direkt auf »SPRG JMP 0000« folgen und der JMP-Befehl bei SPRG in JSR umgewandelt würde.



Damit soll das Thema »Selbstmodifikation« abgeschlossen sein. Die vorgestellten Programmiertechniken bieten fast unbegrenzte Möglichkeiten, hier konnte ich nur einen kleinen Überblick geben, welcher aber für fortgeschrittene Programmierer ausreicht.

11. Mehr über relative Adressierung

So wie wir schon die Tücken der Zeropage-Adressierung zumindest teilweise beseitigen konnten, wollen wir uns mit der in vergleichbarer Weise leistungsstarken Relativ-Adressierung auseinandersetzen.

a) So vermeidet man JMP

Oft muß eine Stelle im Programm angesprungen werden, ohne daß erst eine Bedingung geprüft wird. Diese Stelle ist nicht selten weniger als 128 Byte vom Sprungbefehl entfernt, könnte also relativ adressiert werden.

Dennoch ist es in vielen Fällen möglich, einen Branch-Befehl – obwohl diese Befehle eine Bedingung (C=0..) prüfen – zu verwenden.

Beispiel:

```
7050    BNE 7040
7052    JMP 708A
```

Kann ersetzt werden durch:

```
7050    BNE 7040
7052    BEQ 708A
```

da bei 7052 in jedem Fall das Z-Flag = 0 ist (dafür sorgt der Abfang-Befehl BNE) und somit immer verzweigt wird.

Man könnte den BEQ-Befehl als »Pseudo-Verzweigungsbefehl« bezeichnen, da die Bedingung gar nicht überprüft werden müßte (sie ist sowieso erfüllt).

Der Branch-Befehl übertrifft den JMP-Befehl deutlich an Effektivität, da ein Byte weniger verbraucht wird.

Im übrigen ist auch bei

```
7050    BVS 7040
7052    CLV
```

der CLV-Befehl überflüssig, solange vor 7052 der Befehl von 7050 verarbeitet wird.

b) Zugriff auf Befehle in »Umgebung«

Unter »Umgebung« wollen wir den Bereich um einen Programmteil verstehen, der über relative Adressierung angesprochen werden kann. Da in diesem oft ähnliche Befehlsfolgen stehen wie im anderen Programm, läßt sich hier durch gezielten Zugriff auf die »Umgebung« der Speicherplatzbedarf senken.

Beispielsweise stehen an vielen Stellen im Programm RTS-Befehle. Diese werden, wenn ein Unterprogramm verlassen werden soll, manchmal durch einen Branch-Befehl angesprungen.

```
X1     RTS       ; Ende eines im Speicher voraus-
                   gehenden Unterprogramms
UP     ......    ; Unterprogramm
TEST   BEQ X2    ; Unterprogramm verlassen, falls Z=0
       ......    ; andernfalls weiteres Programm
X2     RTS       ; Ende des Unterprogramms
```

Wenn X1 von TEST aus relativ adressiert werden kann, können wir folgendermaßen ein Byte sparen:

```
X1     RTS
UP     ......
TEST   BEQ X1    ; nach X1 springen, wo auch ein RTS
                   steht
       ......
X2     RTS       ; wird nicht mehr benötigt
```

Noch ein Beispiel aus dem Basic-Interpreter. Bei Adresse $AF08 stehen zwei Befehle, die einen SYNTAX ERROR erzeugen.

Nun gibt es im Basic-Interpreter unzählige Stellen, an denen ein SYNTAX ERROR aufgerufen werden muß. Deshalb steht dort nur »JMP $AF08«. Diese Stellen werden bei Bedarf relativ adressiert, so daß nicht an jeder Stelle, an der ein SYNTAX ERROR aufgerufen wird, der Befehl »JMP $AF08« stehen muß.

Zur Übung könnten Sie noch versuchen, im Programm Tabellen-Beispiel (Listing 11) die Menüroutine (insbesondere die Routinen HOME, DOWN, UP, EXEC), in der beispielsweise wiederholt STX MPT steht, durch Zugriff auf »Umgebung« zu optimieren. Besonders hilfreich dürfte es sein, zunächst statt Branch-Befehlen JMPs einzusetzen und dann zu überlegen, inwieweit die JMPs durch Branches ersetzt werden können, weil zum Beispiel nach »LDX # 0« das Z-Flag immer gesetzt ist etc.

12. Puffer-Technik

In der Computerei fällt der Begriff »Puffer« sehr häufig. Beim C 64 gehören der Kassetten- und der Tastaturpuffer gemeinhin zu den bekanntesten Puffern. Statt »Puffer« kann man auch Zwischenspeicher sagen. Puffer dienen nämlich immer als Zwischenspeicher.

Zunächst wollen wir klären, was zu einem Puffer gehört.

a) Was benötigt ein Puffer?

– Pufferspeicher

Selbstverständlich muß ein Puffer einen bestimmten Speicherbereich belegen, in dem die Werte zwischengespeichert werden.

Ebenso muß die maximale Puffergröße festgelegt werden, damit geprüft werden kann, ob sich der Puffer schon angefüllt hat. Beim Kassettenzugriff werden vorerst alle Byte, die auf die Kassette sollen, im Puffer (ab $033C) zwischengespeichert. Ist dieser Puffer voll, würde er beim nächsten Byte, das er aufnehmen soll, überlaufen (das heißt, die maximale Puffergröße überschreiten). Deshalb wird dann Byte für Byte der Puffer entleert, indem die Bytes auf Kassette geschrieben werden. Jedes Byte, das auf Kassette geschrieben wurde, belegt keinen Speicher mehr im Puffer, so daß der Puffer wieder aufnahmefähig ist.

Damit das Programm, das den Puffer verwaltet, auch weiß, aus welcher Adresse im Puffer es sich das nächste Byte holen soll beziehungsweise wo im Puffer das nächste Byte abgelegt werden soll, gibt es noch einen

– Pufferzeiger

Auf englisch heißt er »BUFFER-POINTER«, woher auch die Abkürzung »B-P« beim Floppy-Befehl zur Manipulation des Pufferzeigers stammt.

Dieser Pufferzeiger kann mit dem Stapelzeiger verglichen werden. Auf keinen Fall ist er mit dem

– Puffervektor

zu verwechseln, der die Startadresse des Pufferspeichers beinhaltet. Ein Puffervektor ist nicht unbedingt erforderlich, erhöht aber die Flexibilität.

Damit wären die Fachausdrücke im Zusammenhang mit Puffern geklärt.

b) Wann verwendet man Puffer?

Puffer dienen in der Regel als Zwischenspeicher, wie zum Beispiel der Basic-Eingabepuffer (ab $0200).

Im Fall des Tastatur- oder Diskettenpuffers aber sind die Puffer als Verbindungsstelle zwischen zwei parallel arbeitenden Programmen beziehungsweise Peripheriegeräten vorgesehen (interruptgesteuerte Tastaturabfrage/Hauptprogramm im Computer, DOS/Betriebssystem des Computers).

Die Puffer sind in diesen Fällen ein Bereich, auf den zwei (quasi-) parallel arbeitende Programme zugreifen.

Bei Computern, die ein wirklich starkes Multitasking bieten (wie der Commodore Amiga) finden Puffer weitaus mehr Verwendung als beim C 64, der nur einen quasiparallelen Ablauf ermöglicht.

Daher werden bei ihm Puffer hauptsächlich im I/O-Bereich verwendet, zum Beispiel bei Druckern, Datasette, Floppy, Tastatur etc. (I/O = Input/Output = Eingabe/Ausgabe).

13. Pass-Technik

a) Begriffserläuterung

Der Begriff »Pass« wurde schon mehrfach im 64'er erläutert (unter anderem Ausgabe 7/85, Seite 51).

Am einfachsten kann der Begriff als »Schritt beim Programmablauf« verstanden werden. Mit »Schritt« ist hier nicht ein einzelner Befehl, sondern ein größerer Block im Programm gemeint.

Wenn ein Programm in 3 Passes (Durchläufen) arbeitet, heißt dies, daß 3 Schleifen hintereinander abgearbeitet werden, die alle eine Teilaufgabe erfüllen, die in Verbindung mit den anderen Passes erst eine größere Aufgabe (zum Beispiel eine Assemblierung) ausfüllen kann. Jeder einzelne Pass führt eine bestimmte Tätigkeit aus, die für das Funktionieren der darauffolgenden Passes unbedingt erforderlich ist. Pass 1 wirkt also wie eine Initialisierung von Pass 2 etc.

Komplexe Programme in Schritte (Passes) zu gliedern, gehört zu den Grundregeln des strukturierten Programmierens.

b) Beispiele von Anwendungen der Pass-Technik

Besonders umfangreiche Programme wie Assembler (Hypra-Ass), Compiler (Austro-Speed) und Interpreter (Comal) sind immer in mehrere Passes eingeteilt.

So erfolgt bei den meisten Assemblern im ersten Pass ein Syntax-Check und das Anlegen der Symbol-Tabelle. Erst im zweiten Pass wird der Objektcode generiert, wobei die bereits erstellte Symboltabelle benötigt wird.

14. Diverse Tips zur optimalen Speichernutzung

Mit übermäßig viel RAM ist der C 64 bestimmt nicht gesegnet. Bei vielen Anwendungen (zum Beispiel Datenverarbeitung) braucht man auch das letzte Byte.

Sie werden nun mehrere Tips erhalten, wie man den wenigen vorhandenen Speicher möglichst sparsam verwenden kann.

Zu den speicherplatzaufwendigsten Einrichtungen gehören die Puffer. Der Kassettenpuffer beispielsweise belegt den RAM-Bereich $033C - $03FB, auf den man somit oft verzichten muß.

Hier wollen wir einfach den Kassettenpuffer in den Bildschirmspeicher (ab $0400 in Normaleinstellung) verlegen.

```
LDA #<$400
LDY #>$400
STA $B2
STY $B3
```

Da der Bildschirm beim Kassettenbetrieb ohnehin abgeschaltet wird, fällt dies nicht auf. Nach dem Kassettenbetrieb sollte man aber den Bildschirm unverzüglich löschen.

Ebenso kann man andere Puffer, für die es einen Vektor gibt, problemlos nach $400 verlegen, sofern sie nicht größer als 1000 Byte sind.

Ein Problem für sich stellt das RAM ab $E000 (also unter dem Betriebssystem!) dar. Diesen Speicher kann man nur durch Bank-Switching nutzen, wobei man noch auf das Betriebssystem verzichten muß, solange der $E000-Bereich auf RAM geschaltet ist.

Hier können wir uns zunutze machen, daß der VIC auch ohne Ändern des Prozessor-Ports (Adresse $0001) auf diesen RAM-Bereich zugreifen kann. Für Grafikbilder oder einen geänderten Zeichensatz ist der $E000-Bereich bestens geeignet.

Oft wird der $E000-Bereich zur Ablage verschiedener Daten verwendet, auf die nicht andauernd zugegriffen werden muß.

Man könnte aber auch das Betriebssystem ins RAM ab $E000 kopieren und diejenigen Bereiche, in denen nicht benötigte Routinen stehen (zum Beispiel für Kassettenbetrieb) einfach überschreiben. Dies ist dann sinnvoll, wenn nur ein paar Byte im $E000-Bereich gebraucht werden. Außerdem ist eine gute Kenntnis des C 64-ROMs erforderlich.

Nun wollen wir noch besprechen, wie der Speicherplatzbedarf eines Programms niedriggehalten werden kann. Dazu wurde im Laufe des Kurses schon einiges gesagt (Unterprogramme statt Makros verwenden etc.).

Jedes Programm benötigt eine Menge Flags. Meist belegt ein Flag genau 1 Byte, für dessen Inhalt es oft nur zwei mögliche Werte gibt: einen für »JA« und einen für »NEIN«.

Für diese primitive Unterscheidungsform genügt aber auch 1/8 Byte, also ein Bit.

Wenn Sie sich das 64'er Extra in der Ausgabe 10/85 ansehen, werden Sie feststellen, daß fast jedes VIC-Register mehrere Funktionen hat, weil jedem Bit eine eigene Bedeutung zukommt. Würde der VIC hier statt auf Bits auf Bytes zugreifen müssen, wäre er

1. langsamer und
2. würde der Speicherplatzaufwand für die Register sich vervielfachen.

Man sollte also bei Flags jedem Bit eine Bedeutung geben und nur die Bits prüfen:

```
BIT FLAG
```


Danach ist das N-Flag gesetzt, falls das 7. Bit im FLAG gesetzt ist, und das V-Flag, falls das 6. Bit gesetzt ist. Die übrigen Flags erhält man über das Z-Flag im Prozessor-Status-Register mit Hilfe des Akkus. Angenommen, man möchte testen, ob Bit 0 im Flag gesetzt ist oder nicht, dann macht das folgendes Programm:

```
LDA #01
BIT Flag
BNE ???        ; (Bit gesetzt)
.
.
.
; (Bit nicht gesetzt)
```

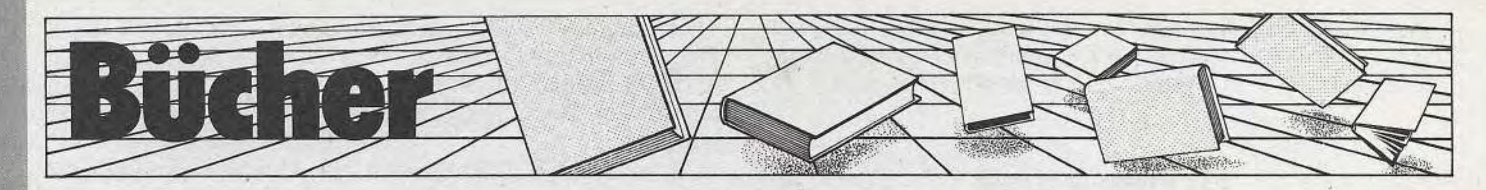
Der Bit-Befehl ANDet den Inhalt des Akkus mit dem Inhalt der Speicherzelle »Flag«. Möchte man Bit 1 testen, so ist der Befehl LDA #01 zu ersetzen durch LDA #02 und so weiter.

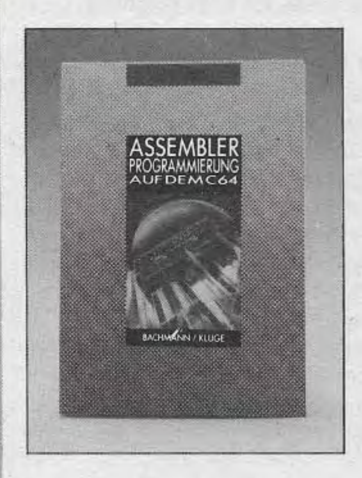
Durch Selbstmodifikation können Flags bekanntlich vermieden werden. Aber auch sonst bietet die Selbstmodifikation die Möglichkeit, Speicherplatz zu sparen: die Steuerung einer Sprungtabelle belegt mit Selbstmodifikation weniger Speicher als ohne.

Auch die »Wegwerfmethode« ist sehr vorteilhaft. Programmteile werden einmal abgearbeitet und dann (zum Beispiel durch Nachladen) überschrieben.

Damit hätten wir unseren Kurs abgeschlossen. Ich hoffe, daß er Ihnen etwas Spaß gemacht hat und Sie einige interessante Informationen herausholen konnten. Sie sollten sich jedoch darüber im klaren sein, daß einige der hier vorgestellten Methoden die Lesbarkeit eines Assembler-Listings einschränken können. Also, verzichten Sie, wenn nicht unbedingt notwendig, auf allzu trickreiche Programmierung. Falls Sie noch Fragen oder Probleme haben (vielleicht erst wegen diesem Artikel), dann schreiben Sie doch einfach.

(Florian Müller/tr)





Assembler-programmierung auf dem C 64

Welcher Einsteiger in die Welt der Computer hat nicht schon voller Neid und Bewunderung auf die zahlreichen Assemblerlistings geblickt?

Die Zeiten sind jetzt vorbei. Mit diesem Buch ist es den Autoren gelungen, jedem Einsteiger den Sprung in die Assemblerprogrammierung zu ermöglichen. Die dazu benötigten mathematischen und schaltungstechnischen Kenntnisse werden im ersten Abschnitt des Buches genau besprochen. Kenntnisse über das Innenleben Ihres Computers, die von großer Bedeutung für die Programmierung in Assembler sind, werden in einem eigenen Kapitel besprochen.

Da der Umgang mit dem Handwerkszeug des Maschinenspracheprogrammierers, nämlich dem Assembler, Disassembler und dem Monitor, den meisten fremd oder ungewohnt ist, richten die Autoren ihr besonderes Augenmerk auf den Umgang mit diesen Programmen. Nach dem sorgfältigen Durcharbeiten dürfte es jedem gelingen, selbst ein kleines Programm zu entwickeln oder eines der zahlreichen Beispielprogramme zu verbessern.

Es werden im Verlauf des Buches weitere fertige Problemlösungen vorgestellt, die oft in selbsterstellte Programme übernommen werden können. Um den Anwender beim selbständigen Arbeiten zu unterstützen, bietet das Buch einiges an Tabellen, insbesondere zum Befehlssatz des Prozessors, zur Umrechnung von Hex nach Dezimal und zur Ansteuerung der Interfacebausteine. Man findet sogar eine herausnehmbare Taschentabelle.

Zusammenfassend bleibt festzustellen, daß man für den Preis von 29,80 Mark ein didaktisch hervorragend aufgebautes Buch erhält, dessen Gesamteindruck noch dadurch verbessert wird, daß zum Buch eine Diskette (49 Mark), mit allen Programmen sowie einem Assembler/Monitor, angeboten wird.

(Udo Reetz/ev)

Info: Walter Bachmann, Norbert Kluge: Assemblerprogrammierung auf dem C 64, Westermann-Verlag, 270 Seiten, ISBN 3-14-138813-X, Preis 29,80 Mark.



Erfolgreicher mit dem VC 64 arbeiten

Mit diesem Buch wird der Autor all diejenigen C 64-Besitzer erreichen, die sich gerade überlegen, wie sie den Sprung »ins Eingemachte«, nämlich die Programmierung in Maschinensprache schaffen sollen.

Dieses Werk berücksichtigt nun die ganze Komplexität des Themenbereiches und widmet sich diesem von der Pike auf.

Der Verfasser, selbst Diplomingenieur, beginnt das Buch mit der Vermittlung von Grundlagen der Digitaltechnik, beschreibt die Funktion verschiedener Zahlensysteme und gibt als Abschluß des ersten Teils des Buches einen Überblick über das grundlegende Funktionsprinzip eines Mikrocomputers, um dann speziell auf den Prozessor 6502 und seinen Befehlssatz einzugehen.

Der zweite Teil des Buches beschäftigt sich dann im besonderen mit dem C64 und seinen Eigenheiten. In gesonderten Kapiteln wird der Speicheraufbau und die Speicherverwaltung dem Leser genauestens dargelegt, im folgenden Abschnitt wird das Augenmerk besonders auf die Interfacebausteine des C64 gerichtet. Es werden die einzelnen Register und deren Programmierung sehr genau beschrieben.

Im dritten Teil werden einige kleine, aber nützliche Programme besprochen, die den Anwender auch zu selbständigen Arbeiten anregen sollen. Dieser Teil des Buches ist gerade für den Einsteiger sehr empfehlenswert, hat er hier doch die Möglichkeit, seine bereits vorhandenen Kenntnisse durch Übung zu erweitern.

Des weiteren findet man über das ganze Buch verstreut zahllose Tabellen, die sowohl eine Hilfe für den Anfänger als auch ein Nachschlagewerk für den Routinier darstellen. Im Anhang gibt es dann noch eine ausführliche Liste der ROM-Routinen mit hex.- und dezimalen Adressen und eine Umrechnungstabelle von Hexadezimal nach Dezimal und umgekehrt.

Alles in allem kann man feststellen: Sowohl für den Einsteiger als auch für den geübten Basic-Programmierer ist dieses Buch sehr gut geeignet, um sich alle nötigen Grundlagen im Umgang mit der Maschinensprache anzueignen.

(Udo Reetz/ev)

Info: Franz Wunderlich: Erfolgreicher mit dem VC64 arbeiten, Franzis' Verlag, 192 Seiten, ISBN 3-7723-7781-5, Preis 38 Mark.



Computerspiele und Wissenswertes – Commodore 64

Vorweg muß gesagt werden: Der Titel dieses Buches ist irreführend. Der Spielefreak, der hier nach einer Bereicherung seiner Listingsammlung sucht, muß ebenso enttäuscht werden wie der Einsteiger auf der Suche nach Basisinformationen. Und höchstwahrscheinlich wird der fortgeschrittene Programmierer dem Buch wegen seines banalen Titels wenig Aufmerksamkeit schenken. Dabei ist gerade für den letzteren dieses Werk interessant. Aus dem Untertitel »Nützliche Maschinenprogramme zum Eintippen« geht schon eher hervor, um welche Art Computerbuch es sich hier handelt: Es wird auf 168 Seiten eine Sammlung von Standard-Maschinenprogrammen angeboten, die im Program-

mieralltag immer wieder gebraucht werden. Vornehmlich soll der fortgeschrittene Assemblerprogrammierer angesprochen werden, der bei größeren Aufgaben schnell und zuverlässig auf Standardlösungen zurückgreifen muß und dabei nicht jedesmal wieder bei Adam und Eva beginnen will.

Diese Publikation ist somit als reines Nachschlagewerk gedacht. Dementsprechend kurz gehalten sind die Erläuterungen zu den verschiedenen Sequenzen. Profis werden schnell das Wesentliche finden, ohne über unnötigen Ballast zu stolpern.

Die Programmbibliothek reicht von einfachen Problemen wie zum Beispiel der Ausgabe von Zahlen und Text oder der Behandlung des Dateisystems bis hin zu speziellen Problemen wie Terminalsimulation oder Verbindung zwischen einem Atari und einem C 64.

Alles in allem kann man sagen, daß sich die Programme problemlos auf eigene Bedürfnisse umstricken lassen.

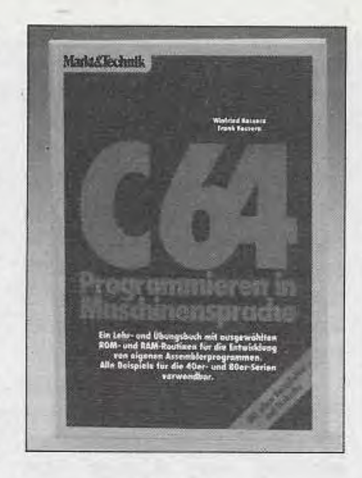
Positiv aufgefallen sind vor allem die Routinen zu Programmierung hochauflösender Grafiken sowie zur Anwendung der RS232-Schnittstelle.

Hingegen ist das Kapitel über die Erweiterung des Basic-Befehlssatzes reichlich mager ausgefallen, Anwendungsbeispiele und Programmvorschläge fehlen hier ganz. Es wäre sicher noch wünschenswert gewesen, zugunsten der Assembler-Neulinge das eine oder andere Grundlagenkapitel anzufügen wie zum Beispiel die Nutzung der ROM-Routinen oder das Arbeiten mit Label etc.

Fazit: Dieses Buch bietet dem eingefleischten Programmierer ein schnelles und zuverlässiges Handwerkszeug und ist in Verbindung mit anderen Büchern sicher auch für Anfangssemester interessant.

(Matthias Rosin/ev)

Info: Computerspiele & Wissenswertes/Commodore 64, Markt&Technik, 166 Seiten, ISBN 3-922120-62-8, Preis 29,80 Mark



C 64 – Programmieren in Maschinensprache

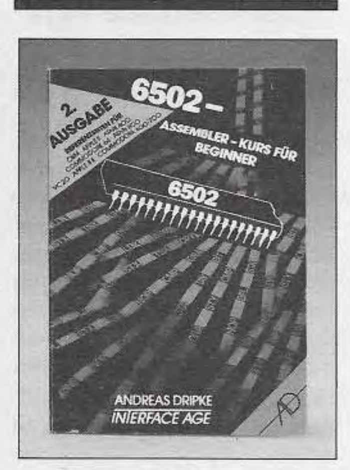
Vorweg gesagt: Dieses Buch eignet sich nicht dazu, sich die Grundbegriffe der Maschinensprache anzueignen. Es spricht vielmehr jenen Leserkreis an, der sich schon gut in Basic eingearbeitet hat und der jetzt mit der fortgeschrittenen Programmierung in Maschinensprache seine Programme optimieren will. Dazu bietet das Buch nicht nur C 64-, sondern auch den Besitzern der CBM 40XX/80XXer-Serie zahlreiche, gut ausgeführte, fertig ausgetestete Programme aus allen Anwendungsbereichen wie zum Beispiel Grafik, Floppy und Dateiverwaltung. Der Anwender muß die für ihn zutreffende Problemlösung nur noch von der ebenfalls mitgelieferten Diskette laden und in sein Programm einbinden. Dadurch werden lästige Tippfehler und die damit verbundene, oft langwierige Fehlersuche vermieden.

Das Buch ist aber nicht nur als fertige Softwarebibliothek zu betrachten, es enthält zudem zahllose Beispiele und Aufgaben, die den Sinn haben, das in den Programmen vermittelte Wissen weiter auszubauen und zu vertiefen. Ganz hervorragend werden die Ein- und Ausgaberoutinen und sämtliche Arithmetikfunktionen erklärt, die so in keinem anderen Buch zu finden sind. Durch die Verwendung der Interpreter-Routinen wird der Umgang mit Zahlen kinderleicht. Weiterhin findet man im Anhang eine Tabelle mit dem vollständigen Befehlssatz des 6502; jeder Befehl ist mit einem Beispiel versehen, das die Wirkung auf die Flags gut darstellt. Genauso ausführlich werden die wichtigsten Einsprungadressen in das Betriebssystem beschrieben. Somit erhält sowohl der weniger versierte als auch der Profi ein immer zu gebrauchendes Nachschlagewerk. Auch bei häufigem Gebrauch wird das Buch seinen Leser nicht durch lose Seiten enttäuschen, denn das Ganze ist von einem stabilen Einband umgeben.

Ein Mangel fällt trotz der vielen Pluspunkte dennoch ins Auge: Der Leser muß im Besitz eines Assemblers/Disassemblers sein, da ein solches Programm nicht mitgeliefert wird. Dieser Minuspunkt wird aber durch die im Preis enthaltene Diskette wieder weitestgehend ausgeglichen.

Für einen Preis von 52 Mark erhält man mit dem Buch und der Diskette ein inhaltstarkes Nachschlagewerk, das seinesgleichen sucht. (Udo Reetz/ev)

Info: W. Kassera, F. Kassera: C 64 – Programmieren in Maschinensprache, Markt&Technik, 328 Seiten, ISBN-3-89080-168-9, Preis 52 Mark.



6502–Assemblerkurs für Beginner

Dieser Assemblerkurs von Andreas Dripke, dem »Vater« von Exbasic Level II, ist eine Einführung in die Assemblersprache des 6502-Mikroprozessors. Auch Besitzer eines C 64 sind mit dem Buch angesprochen, denn der 6510-Mikroprozessor ist voll kompatibel zum 6502.

Schon das Äußere des Buches weicht von der Norm ab. Es ist kein starres, gebundenes Buch, sondern es macht mehr einen ringbuchartigen Eindruck. Auch die Seitennumerierung ist anders als bei anderen Büchern, denn Andreas Dripke hat in seinem Buch die Seiten kapitelweise durchnumeriert. Das hier besprochene Buch bezieht seine Angaben auf einen ganz bestimmten Assembler (genannt T.EX.AS = Terminal Extended Assembler), welcher ebenfalls aus dem Hause Dripke stammt. Parallel zur Anschaffung des Buches empfiehlt sich die Anschaffung von »T.EX.AS«, obwohl der Lehrgang auch mit anderen Assemblern erfolgreich durchzuarbeiten ist.

Das Buch ist für die Theorie als auch für die Praxis geschrieben, so daß man es am besten am Computer sitzend bearbeitet. Die zu lernenden Assemblerbefehle werden in kleinen Programmen vorgestellt, so daß die Wirkung aller Befehle sofort ausprobiert werden kann.

Zwischen diesen mehr praktisch orientierten Kapiteln sind andere mehr theoretischer Art eingestreut, in welchen von Bits und Bytes erzählt wird, der Speicheraufbau erläutert wird oder die verschiedenen Adressierungsarten in der Assemblersprache besprochen werden. Mit theoretischen Kapiteln ist allerdings nicht gemeint, daß diese Kapitel langweilig sind. Auch sie sind, wie das ganze Buch, in einer lockeren, leicht verständlichen aber doch korrekten Sprache geschrieben. Und hat man wirklich einmal etwas nicht verstanden, so kommt am Ende eines jeden Kapitels noch einmal eine Zusammenfassung, in welcher das Wichtigste der letzten Seiten wiederholt wird, neu erlernte Befehle noch einmal aufgelistet sind.

Ein Wort noch zum letzten Kapitel des Buches – Betriebssystemroutinen. Dieses für einen Assemblerprogrammierer äußerst wichtige Kapitel wurde, bedingt durch die Konzeption des Buches, für mehrere Computer geeignet zu sein, nur recht allgemein abgehandelt. Als C 64-Besitzer würde man sich hier mehr Informationen über seinen Computer wünschen, ohne gleich noch ein weiteres Buch anschaffen zu müssen. Ein Wunsch, der vielleicht durch einen computerspezifischen Anhang einmal in Erfüllung gehen könnte? (aw)

Info: Andreas Dripke: 6502-Assemblerkurs für Beginner, Interface Age Verlag, ISBN 3-88986-000-1, Preis 29,80 Mark



64 Intern

Obwohl Data Becker in den letzten Jahren den Computer-Literaturmarkt mit einer Vielzahl von Werken überschwemmt hat, gehört eines der ersten zu den besten. »64 Intern« war zu einer Zeit, als kaum jemand den Commodore kannte (beziehungsweise kaufte, weil er zu teuer war), das einzige Buch, das den mit seinem Handbuch alleine gelassenen C 64-Besitzer von seinem Informationsdefizit befreien konnte.

Ähnlich einem Systemhandbuch aufgebaut, soll »64 Intern« durch gute Beschreibung der Hardware und ihrer Programmierung ein ständiger Begleiter beim Programmieren sein. Diesen Anspruch unterstreicht das kommentierte ROM-Listing (in hexadezimaler Form) am Ende des Buches. Die wichtigsten Kapitel dieses Buches sind:

— Hardware (CPU, Speicherbelegungspläne, User-Port, Expansion-Port)
— Tonprogrammierung (Der SID, Register, A/D-Wandler, Synthy 64)
— Grafik (Der VIC, Registerbeschreibung, Betriebsarten, Schnittstellen zum Prozessor,)
— Ein/Ausgabebausteine (Register-Plan, Ports, Timer, die CIAs, Joystickprogrammierung)
— Der Basic-Interpreter (Erweiterung des Basic, Monitor-Programm, wichtige Kernal-Adressen, RS232, serieller Bus)
— Vergleich: VC 20 - CBM - C 64
— ROM-Listing

Ebenso wie das Interface Age Systemhandbuch ist »64 Intern« für den fortgeschrittenen Programmierer nahezu unverzichtbar. Obwohl sich einige Informationen beider Bücher überschneiden, wird gerade durch die unterschiedliche Erklärung in beiden Büchern mancher Sachverhalt erst richtig verständlich. (aw)

Info: Angershausen, Brückmann, Englisch, Gerits: 64 Intern, Data Becker, 4. Auflage 1984, 350 Seiten, Preis: 69 Mark

Das Interface Age Systemhandbuch zum Commodore 64 und VC 20

Mit diesem Buch haben die Profis von Interface Age ein Handbuch für den fortgeschrittenen Programmierer geschaffen. Die Autoren Ralph Babel, M. Krause und A. Dripke haben in dieses mehr als 300 Seiten starke Buch ihre ganzen Erfahrungen aus der jahrelangen Arbeit mit Commodore-Computern einfließen lassen. Die eindeutige Aussage dieses Buches liegt in der Einsicht, daß ein so hervorragender Computer wie der C 64 erst in Assembler richtig programmiert werden kann. Dazu bedarf es aber der Kenntnis jedes einzelnen Bausteins im C 64. Herabsteigend bis auf die Chip-internen Grundvoraussetzungen werden die Bedingungen der Programmierung des C 64 umfassend und leicht verständlich beschrieben. In den vielen ausführlichen Kapiteln räumen die Autoren mit Unklarheiten und Halbwahrheiten über den C 64 auf. Die folgenden Themen werden dabei behandelt:

— Der Basic-Interpreter (Interne Codierung, Token-Tabelle, Binärarithmetik, Darstellung und Ablage von Variablen, USR-Funktion, Steuercode-Tabelle
— Assembler (Assembler-Programm, Befehlsliste, Befehlserklärung, Adressierungsarten)
— Grafik und Farbe (Bildschirm- und Farbspeicher, Zeichengenerator, HiRes-Grafik, Sprites, Interrupt- und Grafikkontrolle, Screen Blanking, Smooth Scrolling, Registerübersicht, Farben, Sprite-Generator, Hardware des VIC)
— Funktionstasten
— Tonerzeugung (Tongeneratoren, Frequenzberechnung, ADSR-Funktion, Wellenformen, Tonerzeugung, A/D-Wandler, Registerübersicht, Hardware des SID)
— Ein-/Ausgabe (Serieller Bus, RS232, CIA-Chip, Portprogrammierung, Timer, Interrupt-Handling, Registerübersicht, Kontrollports, Joystick, Paddles, Lightpen, Datenspeicherung auf Kassette/Diskette)
— Adaption von CBM-Programmen
— Speicheraufteilung (Speicherübersicht, dokumentierte Memory-Map, CPU-Speicherverwaltung, Prozessorport, Banking, Pinbelegung der CPU)
— ROM-Listing (ausführlich kommentiert)

Besonders das ausführlich kommentierte ROM-Listing ist von unschätzbarem Wert, denn kaum ein Assembler-Programm, das nicht auf die Systemroutinen angewiesen ist. Leider sind alle Adressen in dezimaler Schreibweise aufgeführt. Besser wäre es gewesen, die vom Assembler her gewohnte hexadezimale Form zu verwenden. Auch wer das Betriebssystem an seine eigenen Bedürfnisse anpassen möchte, findet hier alle notwendigen Informationen. Dieses Buch trägt zu Recht den Namen »Systemhandbuch«. Kompromißlos richtet sich das Buch an den fortgeschrittenen Programmierer, trotzdem ist es leicht verständlich. (aw)

Info: R. Babel, M. Krause, A. Dripke: Das Interface Age Systemhandbuch zum Commodore 64 und VC 20, Bezug durch den Fachhandel, ISBN 3-88986-001-X, Preis 74 Mark



Das C64 Profihandbuch

Schon beim »Diagonallesen« dieses Buches, das mit 410 Seiten zu den umfangreicheren Werken zählt, fällt eines sofort auf: Hier handelt es sich nicht bloß um »noch ein Nachschlagewerk«, sondern um ein wirkliches Hilfsmittel für den Profi. Die Autoren erheben von vornherein nicht den Anspruch der Vollständigkeit. Dieses Buch setzt hingegen gewisse Grundkenntnisse voraus und bietet auf weiterführender Ebene eine umfassende Informationsfülle. Der Handbuchcharakter wird durch ein sehr umfangreiches Stichwortverzeichnis unterstützt, wie es leider sonst bei Computerliteratur eher unüblich ist.

Der Inhalt gliedert sich in fünf logische Kapitel:
Der erste Abschnitt behandelt eine Auswahl allgemeiner Routinen und Algorithmen, die sich auf jedem Computer realisieren lassen. Die ausholende Besprechung überrascht denjeni-

gen, der die »lexikographische Langeweile« erwartet hatte, die die meisten Handbücher auszeichnet. Die Autoren wählen hier einen Lehrbuchstil, der aber durch seine allgemeinverständlichen Texte, gewürzt mit Grafiken, Zeichnungen und kurzen Listings, trotz der anspruchsvollen Themen überzeugt. Diesem Teil des Buches wird der Profi viele nützliche Tips entnehmen können, denn die Autoren rücken einigen Problemen mit fast wissenschaftlicher Akribie zu Leibe. Positiv fallen in diesem Zusammenhang auch die Verweise auf weiterführende Literatur auf. Behandelt werden beispielsweise Sortierverfahren, Menütechniken, Mischverfahren, Selektieren und vieles mehr.

Kapitel 2 stellt einige ausgewählte Basic-Programme und Maschinenroutinen vor, die an sich noch keine Besonderheit darstellen, die aber wieder durch die gelungene Dokumentierung bestechen. Im zweiten Teil des Kapitels werden einige Maschinenroutinen vorgestellt, die die Autoren später ausführlich zu einer recht ansehnlichen Basic-Erweiterung zusammenbasteln. Die Grafik- und Tonerzeugung wird in diesem Kapitel ebenfalls eingehend demonstriert.

Das dritte Kapitel widmet sich der Beschreibung der I/O-Ports und Kapitel vier bietet einige Tips und Tricks, die eigentlich schon keine mehr sind, weil altbekannt. Sei's drum, der Vollständigkeit wegen gehören auch solche Dinge in ein echtes Profi-Buch.

Kapitel fünf fördert das Innenleben des C64 in Tabellen und Abbildungen zu Tage. Beispielhaft sind auch hier wieder die Erläuterungen und die Übersichtlichkeit.

Schließlich finden sich im Anhang für die Basic-Erweiterung noch ein Basic-Lader und ein Hex-Listing sowie der aus dem 64'er-Magazin übernommene Checksummer als Abtipphilfe für alle Listings.

Fazit: Mit diesem Buch ist ein von der Konzeption her vorbildliches Werk verfügbar, das dem engagierten C64-Anwender nicht nur Nachschlagewerk, sondern auch Lehrbuch sein wird. Wer seinen Computer schon für ausgereizt hält, wird hier eines Besseren belehrt werden und wieder neue Anregungen finden.

(Matthias Rosin/ev)

Info: Hans Lorenz Schneider, Werner Eberl: Das C64 Profihandbuch, Markt&Technik, 410 Seiten, ISBN 3-89090-110-7, Preis 52 Mark



C 64 Computer-Handbuch

Auf der Titelseite heißt es: »Einführung und Referenz für kompetentes Arbeiten«. Und das ist es in der Tat. Das Buch ist eine deutsche Übersetzung und Bearbeitung der englischen Originalausgabe »Programming the Commodore 64« und kann getrost als eines der wenigen Standardwerke zum C 64 gelten.

Auf über mehr als 600 Seiten findet jeder, der sich etwas intensiver mit dem C 64 beschäftigen will, eine Menge Informationen, Wissenswertes, Tips und Tricks, Grundlagen und Hinweise für Profis. Betrachtet man alleine die Kapitel über Basic, wird sogar Spezialisten in dieser Hinsicht das Lesen bestimmt nicht langweilig werden. Man merkt mit jeder Seite, daß dieses Buch von einem wirklichen Könner mit langer praktischer Erfahrung geschrieben wurde, ohne überflüssigen Ballast, konzentriert und doch an wichtigen Stellen ausführlich genug. Das Buch ist in 17 Kapitel aufgeteilt, zusätzlich kommt ein Anhang mit wichtigen Tabellen sowie ein ausführliches Stichwortregister. Einen guten Eindruck von dem Umfang der behandelten Themen vermittelt ein Einblick in das Inhaltsverzeichnis:

Kapitel 1 und 2: Dem Vorwort folgt eine allgemeine Einführung über die Eigenschaften des C 64.
3. C 64/SX 64 Basic zum Nachschlagen. Alle Befehle des C 64 mit vielen interessanten Details.
4. Effektives Programmieren in Basic. Optimierung von Basic-Programmen mit vielen Beispielen.
5. Architektur des C 64. Einführung in die Hardware des C 64, alles über die Ports.
6. C 64/SX 64 Basic für Professionelle. Besonderheiten des Basic und eine Reihe von Dienstprogrammen und Erweiterungen.
7. Einführung in die Maschinensprache des 6510. Umfassende Beschreibung der CPU 6510 mit Beispielen und Problemlösungen.
8. Typische Methoden der C 64 Maschinenprogrammierung. Wie man das Kernel, Basic-Routinen und das RAM unter dem ROM nutzt sowie Ändern von Basic-Befehlen etc.
9. Verbindung von Basic und Maschinencode.
10. Der Befehlssatz der CPU 6510.
11. ROM-Führer. Speicherbelegung und Betriebssystemroutinen.
12. Grafik
13. Ton und Musik
14. Band
15. Diskette
16. Die Spiele-Ports. Joysticks, Drehregler, Grafiktablett, Maus und Lichtgriffel etc.
17. Drucker, Plotter, Modems

Dieses Buch kann jedem wärmstens empfohlen werden, der sich etwas näher mit dem C 64 beschäftigen will, sei es nur in Basic oder auch in Maschinensprache. Ein Handbuch, das garantiert nicht im Regal verstaubt.

(gk)

Info: Raeto West: C 64 Computer Handbuch, TeWi-Verlag, 600 Seiten, ISBN 3-921803-24-1, Preis 66 Mark.

PEEKs und POKEs zum Commodore 64

Dieses Buch erweckt sofort das Interesse, verspricht doch der Titel endlich einmal ein Werk, mit dessen Hilfe man sich durch den »POKE-Wald« des Commodore 64 kämpfen kann, ohne dabei an den Rand des Wahnsinns zu gelangen.

Dieses Buch ist für Anfänger gedacht und beginnt mit einer lockeren Einführung in die Arbeitsweise des Mikroprozessors 6502 und schließlich des gesamten Computers. Da die Informationen sehr ausführlich nahegebracht werden, wird auch der blutigste Einsteiger schon innerhalb kurzer Zeit das Konzept eines Mikrocomputers verstehen.

Nach den Grundlagen beginnt der Autor die wichtigsten Adressen des Speichers zu erläutern und die Funktionsweise anhand von Beispielen ausführlich darzustellen. Das fängt bei der Peripherieverwaltung an, geht über die Grafik und den Ton, bis hin zur Tastatur und schließlich zu Basic und Betriebssystem.

Da praktisch keine Vorkenntnisse verlangt werden und das Buch zudem sehr erfrischend und spannend geschrieben ist, wird auch der Anfänger bei der Lektüre nicht überfordert, und er wird schnell mit den Möglichkeiten seines C 64 vertraut.

Etwas negativ erscheint nur der »Minikurs« für Maschinensprache am Ende des Buches, der eigentlich überflüssig ist, da sich die Adressaten des Buches, die noch ihre Anfangsschwierigkeiten mit PEEK und POKE überwinden müssen, sicherlich nicht in der Lage sehen, auch schon in Maschinensprache einzusteigen.

Insgesamt sicher ein empfehlenswertes Buch, bei dem auch der wichtige Speicherbelegungsplan zum C 64 nicht fehlt.

(Karsten Schramm/ev)

Info: Liesert: PEEKs & POKEs zum Commodore 64, Data Becker 1984, 150 Seiten, ISBN 3-89011-032-0, 29 Mark



Alles über den Commodore 64

Der Band 1 aus der Commodore-Sachbuchreihe ist ein Standardwerk zum C 64. Wie schön wäre es, wenn dieses gute Buch mit jedem C 64 anstelle des mageren Handbuches ausgeliefert würde. Erstmals auf der Hannover Messe 1984 angeboten, erfüllt dieses Programmierhandbuch nun einen lange gehegten Wunsch vieler Programmierer: Der englischsprachige »Programmers Reverence Guide« wurde ins Deutsche übersetzt.

Das fast 500 Seiten starke Programmierhandbuch ist im wesentlichen in drei Abschnitte gegliedert: die Programmierung in Basic, in Assembler und die Beschreibung der Hardware. In allen drei Abschnitten wird auf die besonderen Baugruppen (Grafik, Musik, Ein-/Ausgabe) des C 64 und deren Programmierung eingegangen. Dem Konzept folgend, kein Lehrbuch, sondern mehr ein Nachschlagewerk zu sein, sind die abgedruckten Beispiele relativ kurz gehalten.

Gerade aus der Kürze der Darstellung leitet sich aber der wichtigste Vorteil dieses Buches ab: Es ist sehr leicht, Informationen zu den verschiedensten Problemen zu finden. Sogar bei Fragen zum generellen Konzept des C 64 hilft das Programmierhandbuch, denn am Ende des Buches ist ein kompletter Schaltplan des C 64 eingeheftet.

Es gibt kaum ein Buch zum C 64, das gleichermaßen umfassend informiert und trotzdem leicht verständlich geschrieben ist. Dabei bleibt der Wert des Buches auch mit steigenden Programmierfähigkeiten des Besitzers erhalten, denn viele Kapitel (Bausteinbeschreibung, Assemblerprogrammierung, Z80-Modul) sind erst für den fortgeschrittenen Programmierer wichtig. Das Programmierhandbuch »Alles über den Commodore 64« sollte neben jedem C 64 liegen, auch wenn derzeit noch 59 Mark zusätzlich investiert werden müssen.

(aw)

Info: »Alles über den Commodore 64«, Commodore Sachbuchreihe Band 1, Bezug über Fachhandel, ISBN3-89133-000-6, Preis 59 Mark

Grafik für Profis

Gerade komplexere Programme im Zusammenhang mit Grafik sind in Basic sehr langsam. Da bleibt dann oft nur noch der Wechsel der Programmiersprache, also statt Basic »Grafik in Maschinensprache auf dem Commodore 64«.

Zunächst einmal muß man das Problem »wo speichere ich was« in den Griff bekommen. Das Buch gibt dabei in seinem zweiten Kapitel eine Hilfestellung indem es aufzeigt, wie die einzelnen Register des Grafikchips programmiert werden müssen. Nach und nach werden nun alle Betriebsarten des Video-Chips angesprochen und anhand von Beispielen in Maschinensprache erläutert. Gleichzeitig wurden zum besseren Verständnis auch die entsprechenden Basic-Programme abgedruckt.

Nachdem man nun erfahren hat, wo man seine Grafik im Speicher unterbringen kann, geht es weiter mit der Definition von Sonderzeichen, sowohl einfarbig als auch in Multicolor. Das nächste Kapitel wird von der hochauflösenden Grafik beherrscht. Eine größere Maschinenroutine, die das Setzen und Löschen von einzelnen Punkten erlaubt, bildet dabei die Grundlage. So bereiten auch kompliziertere Grafiken keine Schwierigkeiten mehr.

Aufgrund dieses ausführlichen Teils kommen die Sprites in diesem Buch etwas zu kurz. Der Autor beschränkt sich bei seinen Ausführungen auf die Beschreibung der Register. Interessant wird es dann noch einmal im 6. Kapitel, wo es um die Besonderheiten des Video-Chips geht. Dazu gehören unter anderem das Softscrolling, die Rasterregister und die Beschreibung des Rasterinterrupts. Ein kleiner Anhang, der jedoch nicht sehr übersichtlich ist, bildet den Abschluß.

Dieses Buch kann man allen empfehlen, die von Basic auf Maschinensprache umsteigen wollen, denn durch die Zweigleisigkeit (Maschinenprogramm und dazu die analoge Basic-Routine) wird das Umdenken bei der neuen Form der Programmierung geübt.

(Cristoph Sauer/ev)

Info: Jürgen Hegner, Grafik in Maschinensprache auf dem Commodore 64, IWT-Verlag, 140 Seiten, ISBN 3-88322-051-5, Preis 38 Mark



Ein Leitfaden für System-Programmierer

Das Commodore-Buch, Band 4«, ein Leitfaden für Systemprogrammierer, ist Teil der C 64-Buchreihe von Markt&Technik. Dem Basic entwachsen, strebt so mancher Programmierer nach Höherem. Dieses Buch wendet sich deshalb vor allem an den Personenkreis, der gerade an der Schwelle der Assemblerprogrammierung steht.

Somit widmen sich auch die ersten Kapitel der Beschreibung des Prozessors und seiner Befehle. Sehr übersichtlich werden Register, Adressierungsarten und Darstellungsformen der Assemblerschreibweise erklärt.

Damit der Programmierer die neuen Kenntnisse auch sofort in die Tat umsetzen kann, haben die Autoren H. L. Schneider und W. Eberl in ihrem Buch gleich zwei Assembler und einen Disassembler mitgeliefert. Die erste Assemblerversion ist in Basic geschrieben. Die zweite Version wurde in durch Maschinensprache unterstütztem Basic entwickelt. Selbstverständlich stehen zu jedem Programm ausführliche Beschreibungen und Anleitungen zur Verfügung. Bei sorgfältigem Durcharbeiten der Programmanleitungen lernt der Einsteiger in die Maschinensprache-Programmierung nicht nur eine ganze Menge über die Arbeitsweise eines Assemblers, sondern auch einiges über die Verknüpfung von Basic und Maschinensprache.

Weitere Kapitel widmen sich der Zusammenarbeit von Maschinenprogrammen mit Basic. Sehr gut gelungen ist dabei die Tabelle der Basic-ROM-Routinen. Sie helfen besonders bei der Erstellung eigener Maschinenprogramme, denn durch geschicktes Verwenden dieser Routinen wird viel Platz gespart. Der Leser lernt nebenbei rationell zu programmieren.

Trotzdem ist der Leitfaden für Systemprogrammierer kein Assembler-Lehrbuch, sondern ein für den Assembler-Anfänger lehrreiches und für den Profi hilfreiches Buch, das in keiner Sammlung fehlen sollte.

Der pädagogische Wert dieses Buches wird durch die für 58 Mark erhältliche Diskette mit allen Programmen unterstützt.

(aw)

Info: H. L. Schneider, W. Eberl: Das Commodore 64-Buch, Band 4, Markt & Technik Verlag, Haar b. München, ISBN 3-922120-70-9, Preis: 38 Mark

Checksummer 64 V3

Der Checksummer 64 V3 überprüft jede Basic-Zeile direkt nach der Eingabe, erkennt Fehleingaben und auch Vertauschungen von Ziffern und erspart eine aufwendige Fehlersuche.

Der Checksummer 64 V3 ist ein kleines Maschinenprogramm, das Sie sofort unterrichtet, ob Sie die jeweilige Programmzeile korrekt eingegeben haben.

So gehen Sie vor:

1. Programm abtippen und speichern.

2. Starten mit RUN

3. Nach kurzer Zeit sehen Sie am Bildschirm: CHECKSUMMER 64, CHECKSUMMER AKTIVIERT, AUSSCHALTEN MIT POKE 1,55, ANSCHALTEN MIT POKE 1,53, READY.

4. Anschalten des Checksummer 64 V3 mit POKE 1,53.

5. Test: Geben Sie in einer freien Zeile ein: »1 REM« und drücken die RETURN-Taste. Am Bildschirm oben links sollten Sie die Prüfsumme <63> sehen.

6. Geben Sie ein Listing aus unserem Heft ein. Nach jeder Zeile wird die Zahl, die im Listing in Klammern < > steht, in den Bildschirm eingeblendet. Stimmen die Zahlen nicht überein, so liegt vermutlich ein Eingabefehler vor. **Die Zahl in den Klammern, und auch die Klammern selbst, dürfen beim Abtippen nicht mit eingegeben werden!**

7. Dieser neue Checksummer 64 V3 bemerkt auch Vertauschungen von Zahlen und Buchstaben, aber nicht das Fehlen (oder Hinzufügen) von Leerzeichen.

8. Unsere Basic-Listings enthalten keine Steuerzeichen mehr. Diese werden ersetzt durch Klartext und stehen zwischen geschweiften Klammern. Deshalb sind weder die Klammern noch was dazwischen steht, abzutippen, sondern die in Tabelle 1 aufgeführten Tasten zu drücken. Auf Ihrem Bildschirm erhalten Sie dann wieder die entsprechenden Grafikzeichen (siehe Bild 1 und 2).

9. Alle Grafikzeichen werden ebenfalls ersetzt durch unterstrichene oder überstrichene Großbuchstaben. **Unterstrichene Buchstaben bedeuten, daß Sie die SHIFT-Taste und den angegebenen Buchstaben drücken müssen, überstrichene jedoch die Commodore-Taste mit dem Buchstaben.** Auch hier erhalten Sie am Bildschirm das entsprechende Grafikzeichen und nicht etwa das im Listing erkennbare Zeichen (siehe Bild 1 und 2).

Die Leerzeichen zwischen den einzelnen Basic-Befehlen können beim Abtippen entfallen (ohne Einfluß auf die Checksumme zu nehmen). Dies ist besonders bei speicherkritischen Programmen wichtig. Ebenso müssen Zeilen, die mehr als 80 Zeichen pro Zeile enthalten, mit den bekannten Abkürzungen für die Basic-Befehle (siehe auch das Handbuch zum C 64, Anhang D, Seite 130) eingegeben werden.

Sie können die Programme auch weiterhin ohne den Checksummer eintippen. (F. Lonczewski/gk)

Hinweis: {13 SPACE} bedeutet 13mal die Leertaste drücken

```
9 REM *********************************
10 PRINT"{CLR,11SPACE,RVSON}CHECKSUMMER 64
   V3{RVOFF}"
11 PRINT"{2DOWN,9SPACE}EINEN MOMENT, BITTE
   ..."
12 FOR I=828 TO 864:READ A:POKE I,A:PS=PS+
   A+1:NEXT I
13 IF PS<>5802 THEN PRINT"PRUEFSUMMENFEHLE
   R IN ZEILEN 20-22":END
14 SYS 828:PS=0:FOR I=58464 TO 58583:READ
   A:POKE I,A:PS=PS+A+1:NEXT I
15 IF PS<>16267 THEN PRINT"PRUEFSUMMENFEHL
   ER IN ZEILEN 22-30":END
16 POKE 1,53:POKE 42289,96:POKE 42290,228
17 PRINT"{4DOWN,9SPACE}CHECKSUMMER AKTIVIE
   RT."
18 PRINT"{2DOWN}AUSSCHALTEN : POKE1,55"
19 PRINT"{DOWN}ANSCHALTEN{2SPACE}: POKE1,5
   3":NEW
20 DATA 169,0,133,254,162,1,189,93,3,133,2
   55,160,0,177,254
21 DATA 145,254,136,208,249,230,255,165,25
   5,221,95,3,208,238,202
22 DATA 16,230,96,160,224,192,0,160,2,169,
   0,170,133,254,177
23 DATA 95,240,40,201,32,208,3,200,208,245
   ,133,255,138,41,7
24 DATA 170,240,14,72,165,255,24,42,105,0,
   202,208,249,133,255
25 DATA 104,170,232,165,255,24,101,254,133
   ,254,76,111,228,192,4
26 DATA 48,219,198,214,165,214,72,162,3,16
   9,32,157,1,4,189
27 DATA 212,228,32,210,255,208,12,0,92,72,
   32,201,255,170,104
28 DATA 144,1,138,96,202,16,228,166,254,16
   9,0,32,205,189,169
29 DATA 62,32,210,255,104,133,214,32,108,2
   29,169,141,32,210,255
30 DATA 76,128,164,9,60,18,19
```

© 64'er

Der Checksummer 64 V3 erkennt auch Vertauschungen von Zahlen.

CTRL steht für Control-Taste, so bedeutet [CTRL-A], daß Sie die Control-Taste und die Taste »A« drücken müssen. Im folgenden steht:

Befehl	Taste	Befehl	Taste
{DOWN}	Taste neben rechtem Shift, Cursor unten	{CYAN}	Control-Taste & 4
{UP}	Shift-Taste & Taste neben rechtem Shift; Cursor hoch	{PURPLE}	Control-Taste & 5
{CLR}	Shift-Taste & 2. Taste ganz rechts oben	{GREEN}	Control-Taste & 6
{INST}	Shift-Taste & Taste ganz rechts oben	{BLUE}	Control-Taste & 7
{HOME}	2. Taste von ganz rechts oben	{YELLOW}	Control-Taste & 8
{DEL}	Taste ganz rechts oben	{RVSON}	Control-Taste & 9
{RIGHT}	Taste ganz rechts unten	{RVOFF}	Control-Taste & 0
{LEFT}	Shift-Taste & Taste unten rechts	{ORANGE}	Commodore-Taste & 1
{SPACE}	Leertaste	{BROWN}	Commodore-Taste & 2
{F1} bis {F8}	Funktionstasten	{LIG.RED}	Commodore-Taste & 3
{RETURN}	Shift-Taste & Return	{GREY 1}	Commodore-Taste & 4
{BLACK}	Control-Taste & 1	{GREY 2}	Commodore-Taste & 5
{WHITE}	Control-Taste & 2	{LIG.GREEN}	Commodore-Taste & 6
{RED}	Control-Taste & 3	{LIG.BLUE}	Commodore-Taste & 7
		{GREY 3}	Commodore-Taste & 8

Tabelle 1. Die Steuerbefehle in den Listings

MSE – Abtippen sicher und leicht gemacht

Ähnlich wie der »Checksummer« ist auch der MSE ein Hilfsmittel bei der Eingabe von Listings, diesmal jedoch bei reinen Maschinensprache-Programmen.

Im Gegensatz zum »Checksummer« aber ist die Eingabe nicht ohne den MSE möglich. Der MSE verringert die Tipparbeit um ein Drittel und schließt Fehleingaben vollkommen aus. Außerdem können Sie die Werte blind eingeben, ohne andauernd auf den Bildschirm schauen zu müssen. Dies wird durch akustische Meldungen realisiert.

MSE ist ein Maschinenspracheditor, mit dem ein Vertippen ausgeschlossen ist. Eine abgetippte Zeile wird nur angenommen, wenn sie richtig ist. Eine Checksumme am Ende jeder Zeile prüft, ob die richtigen Werte in der richtigen Zeile an der richtigen Stelle stehen. Wenn nicht, ertönt ein Warnsignal, und man beseitigt den Fehler.

War die Zeile korrekt, erklingt ein Gong, und die nächste Zeilennummer wird ausgegeben. Damit ist also auch »blindes« Eintippen möglich; Sie können sich voll auf den Text konzentrieren.

So arbeitet man mit MSE

Laden und starten Sie MSE. Zuerst wird der Programmname und die Start- und Endadresse erfragt. **Diese Angaben entnehmen Sie dem Kopf des jeweiligen abgedruckten Listings.** MSE meldet sich dann mit der Zeilennummer der ersten Zeile. Wenn Sie die Zeile richtig eingegeben haben, erscheint die nächste Zeilennummer und so weiter bis zum Ende. Zum Schluß wird das fertige Programm mit »CTRL-S« auf Diskette oder Kassette abgespeichert. Dazu sind keine weiteren Angaben mehr erforderlich. Das Programm kann dann ganz normal wieder geladen und gestartet werden.

Wenn Sie nicht alles auf einmal tippen wollen, können Sie jederzeit unterbrechen und den eingetippten Teil mit »CTRL-S« abspeichern. Wollen Sie weiterarbeiten, laden und starten Sie MSE wieder.

Geben Sie auf die Frage nach der Startadresse aber jetzt »L« ein, um Ihr Teilprogramm zu laden. Jetzt können Sie mit »CTRL-N« die Adresse eingeben, an der Sie weitertippen müssen. Wenn Sie sich nicht gemerkt haben, wie weit Sie gekommen sind, geben Sie nach dem Laden »CTRL-M« ein.

Auf die Frage nach der Startadresse antworten Sie mit der Anfangsadresse, die links in der Kopfzeile auf dem Bildschirm steht. Nun wird Ihr Programm aufgelistet. Mit »SPACE« wird das Listen fortgesetzt, mit »STOP« abgebrochen. Das Ende Ihres Programmteils erkennen Sie sehr einfach daran, daß nur noch der Wert »AA« in der Zeile steht. Die Adresse dieser Zeile müssen Sie anschließend mit »CTRL-N« eingeben. Das Programm ist nur mit »STOP/RESTORE« zu verlassen. Speichern Sie aber vorher unbedingt immer Ihren Text ab.

Hinweise zum Abtippen

Vor dem Abtippen oder späteren Wiederladen des MSE-**Laders** müssen Sie unbedingt folgende Zeile eingeben:

POKE 43,1: POKE 44,32: POKE 8192,0: NEW

Den MSE-**Lader** brauchen Sie nur einmal. Nach erfolgreichem Abtippen und Starten mit RUN geht der Lader verloren und es wird das endgültige Programm MSE V1.0 erzeugt. So gehen Sie vor:

Starten Sie das Programm mit RUN. Fehlerhafte Zeilen werden angezeigt und müssen korrigiert werden, bis der Lader zum »READY« durchläuft. Jetzt müssen Sie das fertige MSE-Programm abspeichern. Dazu brauchen Sie nur »RETURN« zu drücken, weil die erforderlichen Angaben schon auf dem Bildschirm stehen. (Kassettenbesitzer müssen in Zeile 343 die letzte Zahl in »1« abändern.) Ab jetzt können Sie »MSE V1.0« direkt, also ohne den DATA-Lader, benutzen. MSE V1.0 wird ganz normal mit »,8« geladen (keine POKEs notwendig).

(N. Mann / D. Weineck / gk)

MSE-Befehle:

DEL	löscht die letzte Eingabe.
CTRL-S	speichert das eingetippte Programm ab.
CTRL-L	lädt ein Programm. Start- und Endadresse werden automatisch ermittelt.
CTRL-M	listet den Speicherinhalt. Abbruch mit STOP-Taste, weiter mit Leertaste.
CTRL-N	erlaubt die Eingabe einer neuen Adresse zum Weitertippen.
CTRL-P	gibt ein MSE-Listing auf dem Drucker aus.

```
100 REM ****************************         <091>
110 REM *                          *         <159>
120 REM *         M S E   LADER    *         <206>
130 REM *                          *         <179>
220 REM ****************************         <211>
230 REM                                      <036>
240 DIM H(75): FOR I=0 TO 9                  <113>
250 H(48+I)=I: H(65+I)=I+10:NEXT             <041>
260 FOR I=2048 TO 3755 : READ A$             <198>
270 H=ASC(LEFT$(A$,1)):L=ASC(RIGHT$(A$,1))   <199>
280 D=H(H)*16+H(L):S=S+D:POKE I,D            <219>
290 A=A+1:IF A<20 THEN NEXT:A=-1             <141>
300 PRINT " ZEILE:";1000+Z;                  <011>
310 READ V :Z=Z+1:IF V=S THEN 330            <218>
320 PRINT"PRUEFSUMMENFEHLER !":STOP          <138>
330 IF A<0 THEN 341                          <221>
340 S=0:A=0:PRINT:NEXT                       <046>
341 PRINT"{CLR}PO43,1:PO44,8:PO45,172:PO46
    ,14                                      <010>
342 POKE 631,19:POKE 632,13:POKE 633,13:PO
    KE 198,3                                 <?49>
343 PRINT"{3DOWN}SAVE"CHR$(34)"MSE V1.0"CH
    R$(34)",8                                <171>
344 END                                      <092>
1000 DATA 00,0B,08,0A,00,9E,32,30,36,31,00
     ,00,00,A2,08,A9,36,85,A4,A9, 1247       <119>
1001 DATA 08,85,A5,A9,00,85,A6,A9,B0,85,A7
     ,A0,00,B1,A4,91,A6,C8,D0,F9, 2888       <054>
1002 DATA E6,A5,E6,A7,CA,D0,F2,A9,36,85,01
     ,4C,00,B0,20,D1,B1,A9,06,8D, 2787       <144>
1003 DATA 21,D0,A9,03,8D,20,D0,8D,86,02,A0
     ,B3,A9,74,20,FF,B1,A0,B3,A9, 2667       <237>
1004 DATA B9,20,FF,B1,A0,00,20,CF,FF,99,01
     ,02,C8,C9,0D,D0,F5,88,F0,D2, 2912       <217>
1005 DATA C0,0F,90,02,A0,0E,8C,00,02,20,EA
     ,B1,A0,B3,A9,CF,20,FF,B1,20, 2323       <013>
1006 DATA 8E,B4,85,FC,85,62,20,8E,B4,85,FB
     ,85,61,20,A7,B4,D0,20,A0,B3, 2864       <199>
1007 DATA A9,E5,20,FF,B1,20,8E,B4,85,60,20
     ,8E,B4,85,5F,20,A7,B4,D0,0A, 2624       <091>
```

Der MSE zum bequemen Abtippen von Maschinenprogrammen

```
1008 DATA A5,61,C5,5F,A5,62,E5,60,90,06,20
     ,43,B3,4C,3A,B0,A9,AA,A0,00, 2379     <167>
1009 DATA 91,FB,E6,FB,D0,02,E6,FC,20,3F,B2
     ,90,EF,4C,FB,B4,A2,02,86,58, 3118     <152>
1010 DATA A9,A6,A0,9D,20,F2,B1,20,E4,FF,F0
     ,FB,C9,30,90,0C,C9,47,B0,08, 2970     <231>
1011 DATA C9,3A,90,0B,C9,41,B0,07,C9,14,D0
     ,0F,4C,0B,B1,20,D2,FF,A6,58, 2322     <121>
1012 DATA 95,F7,C6,58,D0,D2,60,AE,8D,02,F0
     ,26,C9,0C,D0,03,4C,0B,B6,C9, 2685     <057>
1013 DATA 13,D0,03,4C,8B,B5,C9,0D,D0,03,4C
     ,BA,B4,C9,10,D0,03,4C,68,B5, 2282     <225>
1014 DATA C9,0E,D0,06,20,5F,B4,4C,64,B1,4C
     ,92,B0,A5,F9,20,02,B1,0A,0A, 2132     <208>
1015 DATA 0A,0A,85,F9,A5,F8,20,02,B1,05,F9
     ,60,C9,3A,90,02,69,08,29,0F, 1950     <092>
1016 DATA 60,A6,59,E0,08,90,1F,A6,58,E0,02
     ,B0,06,20,D2,FF,4C,8E,B0,C6, 2509     <188>
1017 DATA 59,A0,14,A9,92,20,F2,B1,CA,D0,FA
     ,84,57,68,68,4C,8B,B1,A6,D3, 2891     <197>
1018 DATA E0,08,B0,03,4C,92,B0,20,D2,FF,A6
     ,58,E0,02,90,09,C6,59,20,D2, 2468     <049>
1019 DATA FF,C6,58,D0,F9,4C,8E,B0,48,4A,4A
     ,4A,4A,20,59,B1,68,29,0F,C9, 2419     <035>
1020 DATA 0A,90,02,69,06,69,30,4C,D2,FF,A2
     ,FC,9A,20,D1,B1,20,48,B2,20, 2261     <073>
1021 DATA EA,B1,20,9F,B2,A5,FC,20,4E,B1,A5
     ,FB,20,4E,B1,20,ED,B1,A9,3A, 2860     <148>
1022 DATA A0,20,20,F2,B1,A9,00,85,59,20,8E
     ,B0,20,ED,B1,A4,59,20,EF,B0, 2530     <233>
1023 DATA 91,FB,C8,84,59,C0,08,90,EC,20,10
     ,B2,A9,12,20,D2,FF,20,8E,B0, 2657     <105>
1024 DATA 20,EF,B0,C5,FF,F0,0D,20,43,B3,A9
     ,14,A0,14,20,F2,B1,4C,A2,B1, 2665     <034>
1025 DATA A9,92,20,D2,FF,20,33,B2,20,E0,B2
     ,20,3F,B2,90,9F,4C,8B,B5,A9, 2648     <123>
1026 DATA 93,20,D2,FF,A2,00,A9,03,9D,00,D8
     ,9D,00,D9,9D,00,DA,9D,00,DB, 2476     <237>
1027 DATA E8,D0,EF,60,A9,0D,2C,A9,20,4C,D2
     ,FF,20,D2,FF,98,4C,D2,FF,20, 2965     <160>
1028 DATA E4,FF,F0,FB,60,84,5D,85,5C,A0,00
     ,B1,5C,F0,06,20,D2,FF,C8,D0, 3100     <077>
1029 DATA F6,60,A5,FB,85,5A,A0,00,84,5B,B1
     ,FB,18,65,5A,85,5A,90,02,E6, 2606     <156>
1030 DATA 5B,06,5A,26,5B,C8,C0,08,90,EC,A5
     ,5A,65,5B,85,FF,60,18,A5,FB, 2467     <219>
1031 DATA 69,08,85,FB,90,02,E6,FC,60,A5,FB
     ,C5,5F,A5,FC,E5,60,60,A0,B3, 3106     <183>
1032 DATA A9,FB,20,FF,B1,A0,01,B9,00,02,20
     ,D2,FF,CC,00,02,C8,90,F4,A9, 2692     <098>
1033 DATA 10,ED,00,02,AA,20,ED,B1,CA,D0,FA
     ,A5,62,20,4E,B1,A5,61,20,4E, 2453     <236>
1034 DATA B1,20,ED,B1,A5,60,20,4E,B1,A5,5F
     ,20,4E,B1,A9,9F,20,D2,FF,20, 2575     <038>
1035 DATA EA,B1,24,5E,10,01,60,A9,12,20,D2
     ,FF,A2,28,20,ED,B1,CA,D0,FA, 2646     <161>
1036 DATA A9,92,4C,D2,FF,A5,D6,C9,16,B0,01
     ,60,A9,A0,85,A4,A9,78,85,A6, 2945     <204>
1037 DATA A9,04,85,A5,85,A7,A2,13,A0,27,B1
     ,A4,91,A6,88,10,F9,CA,F0,19, 2671     <208>
1038 DATA 18,A5,A4,69,28,85,A4,90,02,E6,A5
     ,18,A5,A6,69,28,85,A6,90,E0, 2503     <251>
1039 DATA E6,A7,4C,B6,B2,A9,91,4C,D2,FF,A9
     ,0F,8D,18,D4,A9,00,8D,05,D4, 2776     <000>
1040 DATA A9,F7,8D,06,D4,A9,11,8D,04,D4,A9
     ,32,8D,01,D4,A9,00,8D,00,D4, 2413     <126>
1041 DATA A0,80,20,09,B3,A9,10,8D,04,D4,60
     ,A2,FF,CA,D0,FD,88,D0,F8,60, 2914     <240>
1042 DATA A9,0F,8D,18,D4,A9,2D,8D,05,D4,A9
     ,A5,8D,06,D4,A9,21,8D,04,D4, 2385     <119>
1043 DATA A9,07,8D,01,D4,A9,05,8D,00,D4,A0
     ,FF,20,09,B3,A9,20,8D,04,D4, 2250     <078>
1044 DATA A9,00,8D,01,D4,8D,00,D4,60,38,20
     ,F0,FF,8A,48,98,48,18,A0,06, 2179     <175>
1045 DATA A2,18,20,F0,FF,A0,B4,A9,0A,20,FF
     ,B1,20,12,B3,20,E4,FF,F0,FB, 2931     <093>
1046 DATA A2,1D,A9,14,20,D2,FF,CA,D0,FA,68
     ,A8,68,AA,18,4C,F0,FF,0D,0D, 2704     <088>
1047 DATA 0D,20,20,20,20,20,20,20,4D,41,53
     ,43,48,49,4E,45,4E,53,50,52, 1144     <216>
1048 DATA 41,43,48,45,20,2D,20,45,44,49,54
     ,4F,52,20,0D,0D,20,20,20,20, 1023     <038>
1049 DATA 20,20,20,20,56,4F,4E,20,4E,2E,4D
     ,41,4E,4E,20,26,20,44,2E,57, 1128     <206>
1050 DATA 45,49,4E,45,43,4B,00,0D,0D,0D,20
     ,20,20,50,52,4F,47,52,41,4D, 1102     <117>
1051 DATA 4D,4E,41,4D,45,20,3A,20,00,0D,0D
     ,20,20,20,53,54,41,52,54,41, 1073     <095>
1052 DATA 44,52,45,53,53,45,20,3A,20,24,00
     ,0D,0D,20,20,20,45,4E,44,41, 1014     <129>
1053 DATA 44,52,45,53,53,45,20,20,20,3A,20
     ,24,00,92,05,20,50,52,4F,47, 1171     <217>
1054 DATA 52,41,4D,4D,20,3A,20,00,12,20,20
     ,2A,2A,2A,20,46,41,4C,53,43, 1024     <027>
1055 DATA 48,45,20,45,49,4E,47,41,42,45,20
     ,2A,2A,2A,20,20,92,00,0D,0D, 1058     <098>
1056 DATA 2A,2A,2A,20,45,4E,44,45,20,2A,2A
     ,2A,00,13,05,20,20,12,44,92, 920      <148>
1057 DATA 49,53,4B,20,4F,44,45,52,20,12,54
     ,92,41,50,45,0D,00,13,20,20, 1151     <035>
1058 DATA 49,2F,4F,20,2D,20,46,45,48,4C,45
     ,52,00,20,D1,B1,20,48,B2,A0, 1606     <012>
1059 DATA B3,A9,CF,20,FF,B1,20,8E,B4,85,FC
     ,20,8E,B4,85,FB,C5,61,A5,FC, 3207     <251>
1060 DATA E5,62,90,23,A5,FB,C5,5F,A5,FC,E5
     ,60,B0,19,20,A7,B4,D0,14,60, 2860     <112>
1061 DATA 20,A7,B4,F0,0C,85,F9,20,A7,B4,F0
     ,05,85,F8,4C,EF,B0,68,68,20, 2749     <088>
1062 DATA 43,B3,4C,5F,B4,20,CF,FF,C9,4C,D0
     ,09,20,D1,B1,20,48,B2,4C,0B, 2372     <046>
1063 DATA B6,C9,0D,60,A9,00,85,5E,20,5F,B4
     ,20,EA,B1,20,0D,B5,24,5E,30, 2042     <120>
1064 DATA 05,20,E4,FF,F0,FB,20,E1,FF,F0,26
     ,20,9F,B2,24,5E,10,09,20,4E, 2435     <198>
1065 DATA B5,20,0D,B5,20,60,B5,20,33,B2,20
     ,3F,B2,90,D7,A0,B4,A9,28,20, 2190     <207>
1066 DATA FF,B1,20,E4,FF,C9,0D,D0,F9,A9,00
     ,85,5E,A5,61,85,FB,A5,62,85, 3056     <240>
1067 DATA FC,20,E0,B2,4C,64,B1,A5,FC,20,4E
     ,B1,A5,FB,85,FF,20,4E,B1,A9, 3003     <221>
1068 DATA 20,A0,3A,20,F2,B1,A0,00,20,ED,B1
     ,B1,FB,20,4E,B1,C8,C0,08,90, 2566     <070>
1069 DATA F3,20,ED,B1,24,5E,30,03,A9,12,2C
     ,A9,20,20,D2,FF,20,10,B2,A5, 2190     <059>
1070 DATA FF,20,4E,B1,A9,92,20,D2,FF,4C,EA
     ,B1,A9,FF,85,B8,85,B9,A9,04, 3073     <029>
1071 DATA 85,BA,20,C0,FF,A2,FF,4C,C9,FF,20
     ,CC,FF,A9,FF,4C,C3,FF,20,5F, 3315     <189>
1072 DATA B4,A9,80,85,5E,20,4E,B5,20,48,B2
     ,A2,24,A9,2D,20,D2,FF,CA,D0, 2596     <111>
1073 DATA FA,20,EA,B1,20,EA,B1,20,60,B5,4C
     ,C1,B4,20,B8,B5,A6,5F,A4,60, 2812     <015>
1074 DATA A9,61,20,D8,FF,B0,0A,20,B7,FF,29
     ,BF,D0,03,4C,FB,B4,A9,01,20, 2577     <201>
1075 DATA C3,FF,20,68,B6,A0,B4,A9,4F,20,FF
     ,B1,20,F9,B1,4C,FB,B4,20,68, 2921     <237>
1076 DATA B6,A9,37,A0,B4,20,FF,B1,20,F9,B1
     ,A2,08,C9,44,F0,06,A2,01,C9, 2717     <213>
1077 DATA 54,D0,F1,A9,01,A8,20,BA,FF,A0,00
     ,E0,01,F0,1A,A9,40,8D,20,02, 2403     <101>
1078 DATA A9,3A,8D,21,02,B9,01,02,99,22,02
     ,C8,CC,00,02,90,F4,C8,C8,D0, 2182     <127>
1079 DATA 0C,B9,01,02,99,20,02,C8,CC,00,02
     ,D0,F4,98,A2,20,A0,02,4C,BD, 2018     <025>
1080 DATA FF,20,B8,B5,A5,BA,C9,08,90,33,A6
     ,B9,86,57,A9,01,20,C3,FF,A9, 2800     <022>
1081 DATA 60,85,B9,20,C0,FF,B0,28,A5,BA,20
     ,B4,FF,A5,B9,20,96,FF,20,A5, 2911     <053>
1082 DATA FF,85,61,A5,90,4A,4A,B0,13,20,A5
     ,FF,85,62,20,AB,FF,A5,57,85, 2663     <214>
1083 DATA B9,A9,00,20,D5,FF,90,03,4C,A3,B5
     ,86,5F,84,60,A5,BA,C9,01,D0, 2639     <131>
1084 DATA 0A,AD,3D,03,85,61,AD,3E,03,85,62
     ,4C,FB,B4,A9,13,20,D2,FF,A2, 2300     <120>
1085 DATA 1C,20,ED,B1,CA,D0,FA,60, 1230     <214>
```


MSE (Schluß). Dieses Listing können Sie (müssen aber nicht) mit dem neuen Checksummer 64 V3 in diesem Heft eingeben.

Hypra-Ass – Ein Assembler der Spitzenklasse

Machen Sie es wie die Profis. Schreiben Sie Programme in Maschinensprache. Dieser leistungsstarke Makro-Assembler macht es möglich.

Hypra-Ass ist ein rein in Maschinensprache geschriebener Drei-Pass-Makroassembler mit integriertem Editor für den C 64 mit Floppy 1541. Er wird mit LOAD "HYPRA-ASS",8 geladen und durch RUN gestartet. Nach dem Start meldet sich Hypra-Ass mit »break in 0« und »ready«. Alle Basic-Befehle sind nach dem Start noch zu verwenden, bis auf die Befehle LET und FOR, die Variable anlegen. Der Befehl RUN dient jetzt zum Starten der Assemblierung.

Der Quelltext

Der Quelltext wird vom Hypra-Ass-Editor in Basic-Programmzeilen abgelegt. Soweit wie möglich werden unnötige Blanks dabei eleminiert. Für die einzelnen Quelltextzeilen gelten die folgenden Vereinbarungen:

1. Bei der Eingabe einer Zeile wird hinter der Zeilennummer ein Minuszeichen eingegeben.
2. Jede Zeile enthält höchstens einen Assemblerbefehl.
3. Vor einem Assemblerbefehl darf in derselben Zeile höchstens ein Label stehen.
4. Label beginnen direkt hinter dem Minuszeichen.
5. Vor jedem Assemblerbefehl steht mindestens ein Blank.
6. Label und Assemblerbefehl werden durch mindestens ein Blank voneinander getrennt.
7. Ein Label darf nicht allein in einer Zeile stehen.
8. Kommentar wird durch ein Semikolon vom Rest der Zeile getrennt.
9. Reine Kommentarzeilen müssen als erstes Zeichen hinter dem Minuszeichen ein Semikolon haben.
10. Pseudo-Ops (.ba, .eq...) können direkt hinter dem Minuszeichen beginnen.

Beispiele:

```
100  -.ba $C000
110  -initialisierung
120  -; reine Kommentarzeile
130  -               lda $14; Kommentar hinter einem Befehl
140  -marke          ldx $15; mit Label davor
```

Zur bequemeren Eingabe und Bearbeitung des Quelltextes stellt Hypra-Ass im Editor insgesamt 25 Befehle zur Verfügung (Bild 1).

Rechnungen im Quelltext

Hypra-Ass erlaubt die vier Grundrechenarten plus Potenzierung, die logischen Operationen NOT, AND und OR, die Vergleiche »gleich«, »kleiner« und »größer« sowie den Einsatz der Funktionen <(...) und >(...), die das Low- beziehungsweise Highbyte eines Argumentes liefern. Die logischen Operatoren und die Vergleiche werden wie folgt abgekürzt:

!n! = not !a! = and !o! = or !=! = gleich
!<! = kleiner als !>! = größer als

Das Ergebnis eines Vergleiches ist —1, falls wahr, 0, falls nicht wahr: (1!=!2)=0 (1!=!1)=—1

Auch die NOT-Verknüpfung arbeitet wie in Basic: !n!1 = —2. Das Argument in den Low-/Highbyte-Funktionen muß im Bereich $0 \leq$ Argument ≤ 65535 liegen.

Außer Dezimalzahlen sind Hex-Zahlen erlaubt, die durch ein vorangestelltes Dollarzeichen kenntlich gemacht werden: $C000 = 49152 $10 = 16 $a = 10...

Die Hexzahlen können auch in den Basic-Befehlen verwendet werden.

Hypra-Ass-Variable (Label)

Der Wert einer Hypra-Ass-Variablen kann zwischen 0 und $FFFF liegen. Variablennamen können beliebig lang sein, wobei das erste Zeichen des Variablennamens ein Buchstabe sein muß. Weitere Zeichen können Buchstaben, Ziffern oder das Hochkomma sein. Alle Zeichen des Namens sind signifikant.

Im Zusammenhang mit der Verwendung von Makros muß zwischen globalen und lokalen Variablen unterschieden werden. Jede Variable erhält beim Anlegen eine sogenannte Ordnungszahl, die tatsächlich angibt, im wievielten Makroaufruf das Anlegen stattfand. Befinde ich mich in gar keinem Makro, ist die Ordnungszahl entsprechend Null.

Variable mit unterschiedlicher Ordnungszahl sind trotz gleichen Namens nicht gleich. Man kann also davon sprechen, daß Variable gleicher Ordnungszahl lokal sind.

Die Konstruktion mittels Ordnungszahlen dient dazu, Fehler durch doppelte Benutzung von Labeln bei mehrmaligem Aufruf von Makros zu verhindern, indem Makros bei jedem Aufruf sozusagen einen komplett neuen Satz von Labeln erhalten.

Andererseits sind aus einem Makro »herausgesehen« alle Variablen mit anderer Ordnungszahl als im Makro selbst »unsichtbar«. Um aber bequem Makros in Makros aufrufen und bequem Label verwenden zu können, die in mehreren Makros benutzt werden sollen (etwa Betriebssystemroutinen), gibt es die globalen Variablen.

Globale Variable sind, wie der Name schon verrät, im Gegensatz zu den lokalen Variablen unabhängig von der Ordnungszahl überall definiert.

Alle Makronamen sind per Definition global.

Alle Variablen sind bei Hypra-Ass redefinierbar gehalten, das heißt alle Variablen können durch eine Wertzuweisung jederzeit verändert werden.

Eine doppelte Benutzung von Labeln vor Assemblerbefehlen wird jedoch durch einen »Label twice«-Error (Bild 2) geahndet, da dies zu einem falschen Ergebnis der Assemblierung führen würde.

Die Makros von Hypra-Ass

Makros sind meist kürzere Befehlsfolgen, die im Quelltext häufiger vorkommen, und deshalb unter einem Makro zusammengefaßt werden. Zu jedem Makro gehört ein Makroname, mit dem es aufgerufen werden kann. An jedes Hypra-Ass-Makro können beliebig viele Parameter übergeben werden, deren aktueller Wert dann bei der Assemblierung im Makro eingesetzt wird. Makros können bei Hypra-Ass an beliebiger Stelle im Quelltext definiert werden. Alle Makronamen sind global, alle Parameter und makrointernen Label sind lokal. Das heißt verschiedene Makros können durchaus Label beziehungsweise Parameter gleichen Namens verwenden.

Ein Beispiel für ein einfaches Makro:

Es wird immer wieder die Befehlsfolge benötigt, Akkumulator und X-Register mit dem Inhalt zweier aufeinanderfolgender Speicherzellen zu laden. Ein Makro dazu könnte folgendermaßen aussehen:

```
100  -.ma ldax (adresse)
110  -           lda adresse
120  -           ldx adresse+1
130  -.rt
```

Der .ma-Pseudobefehl wird gefolgt von einem Variablennamen, dem Makronamen, und einer Parameterliste in runden Klammern, falls Parameter vorhanden sind. Hier ist es ein Parameter, die Adresse der Speicherzelle, die in den Akku soll. Sind mehrere Parameter vorhanden, werden sie durch Kommata getrennt. In die Parameter setzt der Assembler bei jedem Aufruf den aktuellen Wert, der im Aufruf steht. Rufe ich also ldax (2) auf, so entsteht bei der Assemblierung des Makros die Folge lda 2, ldx3, entsprechend führt der Aufruf mit ldax (label) zu lda label, ldx label+1.

Die Parameterliste darf in der Definitionszeile eines Makros nur aus einer Folge von Variablennamen bestehen, während im Aufruf als aktuelle Parameter beliebige Ausdrücke erlaubt sind. Hinter der Definitionszeile mit dem .ma-Pseudo folgt dann der eigentliche Makroinhalt, das heißt das, was bei einem Aufruf des Makros assembliert werden soll.

Natürlich sind hier nicht nur einfache Befehle wie im Beispiel gestattet. Genausogut können im Makro Verzweigungen und Sprünge ausgeführt werden, es kann bedingt assembliert werden und weitere Makros können aufgerufen werden. Für die Schachtelung von Makros besteht keine Grenze außer der Fassungskapazität des Prozessorstacks.

Als Beispiel — wird ein Makro mit zehn internen Labeln 100 mal aufgerufen, ergibt sich schon für die dadurch erzeugten lokalen Label ein Platzbedarf von genau 7 000 Byte.

Sollte irgendwann der Fall eintreten, daß Label und Quelltext zusammen nicht mehr ins RAM passen, erhalten Sie den »too many labels«-Error (Bild 2). Dies ist allerdings mehr ein theoretischer Fall, denn auch bei der Assemblierung von Hypra-Ass selbst wurden trotz extensiver Benutzung von Labels nicht einmal 500 gebraucht. Sie können aber davon ausgehen, daß Ihnen immer mindestens Platz für 1 170 Label zur Verfügung steht — in den allermeisten Fällen sogar erheblich mehr.

Selbstaufrufe von Makros sind auch nicht verboten. Inwieweit eine solche Konstruktion überhaupt sinnvoll sein kann, bleibt jedem selbst zu prüfen.

Zurück zur Makrodefinition: Jede Makrodefinition muß unbedingt mit dem Pseudo .rt (return) abgeschlossen sein. Trifft der Assembler bei der Abarbeitung eines Makros auf .rt, so heißt das für ihn, die Assemblierung hinter dem Aufruf fortzusetzen.

Vor der .ma und .rt-Anweisung dürfen in derselben Zeile keine Label stehen. Die Makrodefinition selbst wird in Pass 1 und Pass 2 überlesen. Es zählen also nur die Makroaufrufe bei der Assemblierung.

Der Aufruf eines Makros erfolgt durch den Pseudobefehl ..., gefolgt vom Makronamen und der aktuellen Parameterliste in runden Klammern.

Wertzuweisung an Label

Zwei Pseudobefehle stehen zur Verfügung, um Label einen Wert zuzuweisen:

.eq — weist einem Label einen Wert zu, ohne die Ordnungszahl des Labels dabei zu verändern.
.gl — erklärt gleichzeitig das Label als global.

Beide Pseudos werden der eigentlichen Wertzuweisung vorangestellt, so wie LET in Basic:

```
100  -.eq marke = $FFC0
110  -.gl label  = $200
```

Bei der Wertzuweisung an Label ist immer der Bereich einzuhalten in dem ein Labelwert liegen darf (0 bis $FFFF).

Einfügen von Tabellen und Text

Drei Pseudo-Ops erleichtern das Einfügen von Tabellen und Text in den Quelltext. Dies sind:

.by — erlaubt das Einfügen von Bytewerten (Werten zwischen 0 und $FF). Einzelne Bytewerte werden durch Kommata voneinander getrennt. Auch Strings der Länge 1 sind als Bytewerte erlaubt. Beispiel:
100 -.by 0, "a", 123, "x", $fa

.wo — erlaubt das Einfügen von Adressen (Werten zwischen 0 und $FFFF). Mehrere Adressen werden durch Kommata voneinander getrennt. Die Adressen werden in der Folge Low/Highbyte in den Objektcode aufgenommen. Beispiel:
100 -.wo marke-1, label*2-1

.tx — erlaubt das Einfügen von Text in den Quelltext. Die einzelnen Zeichen des Textes werden als ASCII-Code im Objektcode aufgenommen. Beispiel:
100 -.tx "beispieltext"

Überall im Quelltext, wo Bytewerte erwartet werden, etwa bei der unmittelbaren Adressierung, können Strings der Länge 1 verwendet werden. Ein Befehl lda#"a" ist also erlaubt.

Die bedingte Assemblierung

Zur Unterstützung der bedingten Assemblierung bietet Hypra-Ass ein IF/ELSE/ENDIF-Konstrukt und ein IF/THEN-Konstrukt. Außerdem steht ein unbedingter Sprungbefehl zur Verfügung.

.on — entspricht dem IF/THEN von Basic. Hinter .on folgt ein Ausdruck, ein Komma und ein zweiter Ausdruck. Ist der erste Ausdruck wahr, wird zu der Zeilennummer gesprungen, die der zweite Ausdruck angibt. Beispiel:
100 -.on switch !=! 7, 400
Es wird die Assemblierung in Zeile 400 fortgesetzt, wenn switch gleich 7 ist.

.go — ergibt einen unbedingten Sprung zu der Zeile, die der Ausdruck hinter .go angibt. Beispiel:
100 -.go 1000

.if — wird gefolgt von einem Ausdruck. Ist der Ausdruck wahr, wird die Assemblierung hinter der .if-Zeile fortgesetzt, bis
.el — gefunden wird. Daraufhin wird
.ei — gesucht und dahinter die Assemblierung fortgesetzt.

Entsprechend erfolgt die Assemblierung von .el bis .ei, falls der Ausdruck hinter .if falsch ist. .el kann auch fehlen, es wird dann direkt hinter .ei fortgefahren.

Auf eine Schachtelung von IF-Konstrukten wurde wegen des Zwecks der bedingten Assemblierung verzichtet. Beispiel:

```
100  -.if switch !=! 6
110  -            lda#0
120  -.el
130  -            lda#2
140  -.ei
```

Wenn switch gleich 6 ist, erhält man lda#0, sonst wird lda#2 erzeugt. Vor den Pseudos .if, .el und .ei dürfen keine Label in derselben Zeile stehen.

Verkettung von Quelltexten

Mit dem Pseudo .ap (apend) kann ein weiterer Quelltext am Ende des Pass 2 automatisch nachgeladen werden, wobei der Programmzähler aus der vorangegangenen Assemblierung erhalten bleibt.

Hinter .ap muß der Name des nachzuladenden Files in Anführungszeichen stehen.

Eine Besonderheit von Hypra-Ass bildet im Zusammenhang mit verketteten Quelltexten der Pseudo-Opcode .co (common).

Dieser Befehl bewirkt zunächst, daß alle Variablen/Label, die hinter der .co-Anweisung in einer Liste stehen, an den nachgeladenen Teil übergeben werden.

Zweitens bleiben alle Quelltextzeilen bis zur common-Zeile beim Nachladen erhalten. Steht also etwa ein Makro vor der common-Zeile, wird auch das Makro übergeben. Zu beachten ist dabei:

a) Es sollten keine Makroaufrufe im common-Bereich stehen, es sei denn innerhalb eines Makros.
b) Die .ba-Anweisung, die die Startadresse des Objektcodes bestimmt, sollte außerhalb des common-Bereiches liegen, damit nach dem Nachladen nicht wieder mit der gleichen Startadresse assembliert wird.
c) Wertzuweisungen an Label sollten ebenfalls außerhalb des common-Bereiches liegen, um Platz für den nachgeladenen Quelltext zu gewinnen.

Direktes Senden des Objektcodes zur Floppy

Der Pseudobefehl .ob (object), gefolgt vom Filenamen ,p,w in Anführungszeichen, sendet den erzeugten Objektcode direkt zur Floppy.

Geschlossen wird das so erzeugte Objektfile durch den Pseudobefehl .en.

Sollte während der Assemblierung ein Fehler entdeckt werden und das Objektfile nicht schon durch die Hypra-Ass-Fehlerroutine geschlossen worden sein, geben Sie bitte CLOSE 14 ein.

Ausgabe von formatierten Listings

1) **.li 1,3,0** sendet ein formatiertes Listing des Quelltextes unter der logischen Filenummer 1 an das Gerät 3 mit der Sekundäradresse 0 (Bildschirm). Die Parameter hinter .li entsprechen denen des OPEN-Befehls. So ist es auch möglich, mit .li 2,8,2, "test,u,w" das Listing auf eine Userdatei zu leiten und so weiter.

Der .li-Pseudobefehl muß der erste Befehl im Quelltext sein, wenn alle Zeilen gelistet werden sollen. Die Zeilen bis einschließlich .li werden nicht ausgegeben. Die gelisteten Zeilen haben folgendes Format:

c000 a0b0c0: 1000 -marke befehl ;kommentar

Die Steuerung der Formatierung erfolgt mit dem Editorbefehl /t. Bei Zeilen, die Pseudobefehle enthalten, wie .eq... werden keine Adressen und Opcodes ausgegeben.

2) **.sy 1,3,0** sendet am Ende von Pass 2 die sortierte Symboltabelle. Die Formatierung wird hier durch /t3,... gesteuert. Die Labelwerte werden hexadezimal ausgegeben.

Eine Zeile der Symboltabelle sieht dann folgendermaßen aus:

sprungziel = $ffd2

Das Listing des Quelltextes erhält die Kopfzeile »Hypra-Ass Assemblerlisting:«. Die Symboltabelle erhält die Kopfzeile »Symbols in alphabetical order«.

3) **.dp t0,t1,t2,t3** setzt aus dem Quelltext heraus die Tabulatoren
t0 = Tabulator für Assemblerbefehle
t1 = Tabulator für den Kommentar bei der formatierten Ausgabe
t2 = Tabulator für die Anzahl der Blanks, die am

/a 100,10 — Automatische Zeilennumerierung. Hier mit der Startnummer 100 und der Schrittweite 10. Die automatische Zeilennumerierung wird ausgeschaltet, indem man direkt hinter dem ausgegebenen Minuszeichen RETURN eingibt.

/o — Re-New eines Quelltextes, der mit NEW gelöscht wurde, falls der Text nicht anderweitig zerstört wurde.

/d ; /d 100 ; /d —100 ; /d 100— ; /d 100-200
Löschen von Zeilen und Zeilenbereichen. Auch für das Löschen einzelner Zeilen sollte man den /d-Befehl verwenden, da man das Minuszeichen hinter der Zeilennummer doch immer wieder vergißt.

/e ; /e 100 ; /e —100 ; /e 100— ; /e 100-200
Formatiertes Listen von Zeilen und Zeilenbereichen. Label, Assembler-Befehle werden gemäß den Tabulatoren übersichtlich untereinander geschrieben.

/t 0,13 ; /t 1,24 ; /t 2,0 ; /t 3,10
setzt die Tabulatoren T0, T1, T2, T3
T0 = Tabulator für Assemblerbefehle
T1 = Tabulator für den Kommentar bei der formatierten Ausgabe
T2 = Tabulator für die Anzahl der Blanks, die am Anfang einer Ausgabezeile ausgegeben werden
T3= Tabulator für die Symboltabelle

/x — Verlassen des Assemblers. Beim Verlassen des Programms wird ein Reset durchgeführt.

/p 1,100,200 — Setzen eines Arbeitsbereichs (Page). Hier Bereich 1 von Zeile 100 bis 200, beide einschließlich. Bis zu 30 solcher Arbeitsbereiche sind erlaubt. Die Parameter der Arbeitsbereiche werden im Kassettenpuffer abgelegt.

/ziffer(n) — Formatiertes Listen der Page.

/n 1,100,10 — Neu Durchnumerieren einer Page mit Startnummer und Schrittweite.

/f 1,"string" — Suchen einer Zeichenkette in einer Page. Dabei sind im String Fragezeichen als Joker erlaubt. Das Fragezeichen ersetzt ein beliebiges Zeichen. Zu beachten ist jedoch, daß im Quelltext unnötige Blanks entfernt wurden, wie ein Vergleich mit den Befehlen /e und LIST zeigt.

/r 1,"string1", "string2"
Ersetzen von Zeichenketten. String 2 darf nicht leer sein. Überall in der Page wird die Zeichenkette aus String 2 durch die aus String1 ersetzt. Auch beim Ersetzen ist in String 2 das Fragezeichen als Joker erlaubt. Da String 1 leer sein darf, können mit diesem Befehl auch Zeichenketten gelöscht werden.

/u 9000 — Setzen des Quelltextstartes (Programmstartes). Normalerweise ist als Startwert die Adresse 7000 eingestellt. Durch Hochlegen des Startes kann man zum Beispiel einen Monitor in dem nun freien Bereich unterbringen.

/b — Anzeige der aktuellen Speicherkonfiguration. Es wird angezeigt:
a) der normale Quelltextstart 7000 als Merkhilfe
b) der aktuelle Quelltextstart
c) das Quelltextende
d) die Anzahl der noch verbleibenden Bytes für den Quelltext

/l"name" ; /s"name" ; /v"name" ; /m"name"
Kurzform der Befehle LOAD, SAVE, VERIFY und MERGE

/g 8 — Die zugehörige Gerätenummer kann mit diesem Befehl eingestellt werden. Voreingestellt ist das Gerät 8.

Zur Unterstützung des Umgangs mit dem Floppy-Laufwerk 1541 sind drei Befehle implementiert:

/i — Lesen des Inhaltsverzeichnisses von Floppy ohne Verlust des geschriebenen Quelltextes
/k — Lesen des Fehlerkanals
/@ — Übermittlung von Befehlen an die Floppy

Diese drei Befehle entsprechen denen des DOS 5.1.

Auch zur Farbgebung des Bildschirms sind zwei Befehle vorhanden, die die Hintergrund- und die Rahmenfarbe setzen.

/ch 0 — Setzen der Hintergrundfarbe
/cr 0 — Setzen der Rahmenfarbe

Nach erfolgter Assemblierung kann nun die erzeugte Symboltabelle mit zwei Befehlen ausgegeben werden:

/! — Ausgabe in unsortierter Form
/!! — Ausgabe sortiert

Es werden nur Label ausgegeben, die entweder global oder von der Ordnung Null sind.

Beide Dumps können mit der CTRL-Taste verlangsamt und mit der STOP-Taste angehalten werden.

Mit OPEN... und CMD... können die Dumps an andere Geräte gesendet werden.

Als Ergänzung zum Basic-Befehl PRINT, der aufgrund der Tokenbildung nicht alle Labelnamen verarbeiten kann, kann der Befehl ← verwendet werden.

Basic-Funktionen wie PEEK sind nur über den PRINT-Befehl erreichbar. Die Funktionen <(...) und >(...) sind außerhalb des Quelltextes nur durch ← zu verwenden. Mit dem ←-Befehl kann genau wie im Quelltext gerechnet werden.

Bild 1. Die Editorbefehle von »Hypra-Ass«

Anfang einer Ausgabezeile ausgegeben werden.
t3 = Tabulator für die Symboltabelle

4) **.st** beendet die Assemblierung

Am Ende des zweiten Passes wird immer die Meldung »end of assembly« gefolgt von der Assemblierungsdauer in Minuten, Sekunden und Zehntelsekunden ausgegeben. Dahinter folgt die Zeile »base = $XXXX last byte at $YYYY«.

Eine Zusammenfassung aller Pseudobefehle finden Sie in Bild 3.

Hypra-Ass-Editor

Eines der Ziele bei der Entwicklung von Hypra-Ass war es auch, die Editierung von Quelltexten möglichst bequem zu machen. Dazu wurden etliche Funktionen, die im normalen Basic-Editor stets gebraucht, aber nie vorhanden sind, in den Hypra-Ass-Editor eingebaut.

Als Grundlage des Hypra-Ass-Editors blieb dabei der Basic-Editor erhalten.

Ein Hypra-Ass-Quelltext wird also im Prinzip genauso eingegeben wie ein Basic-Programm. **Allerdings muß hinter der Zeilennummer immer ein Minuszeichen eingegeben werden,** das den Beginn der Quelltextzeile bildet. So eingegebene Quelltextzeilen werden als ASCII-Zeilen in den Speicher übernommen. Alle überflüssigen Blanks werden entfernt.

Jede eingegebene Zeile wird sofort nach der Übernahme formatiert ausgegeben, um die Übersichtlichkeit des Quelltextes zu gewährleisten. Eine Tabelle aller Editorbefehle finden Sie in Bild 1.

(Gerd Möllmann/ah)

Zusätzlich zu den Fehlermeldungen, die von Interpreterroutinen wie »illegal quantity« oder »syntax« stammen, gibt Hypra-Ass folgende Meldungen aus:

1. **can't number term** — ein Ausdruck kann von Hypra-Ass nicht berechnet werden. Möglicher Grund kann die falsche Abkürzung eines Operators sein.
2. **end of line expected** — bei der Abarbeitung einer Zeile wurde statt des Zeilenendes etwas anderes gefunden.
3. **no mnemonic** — ein Mnemonic kann nicht identifiziert werden.
4. **unknown pseudo** — ein Pseudo-Op wurde falsch abgekürzt.
5. **illegal register** — ein Assemblerbefehl existiert in der gewählten Adressierungsart nicht mit dem gewählten Register.
6. **wrong address** — ein Assemblerbefehl existiert nicht in der gewählten Adressierungsart.
7. **illegal label** — das erste Zeichen eines Labels war kein Buchstabe.
8. **unknown label** — in Pass 2 wurde ein unbekannter Labelname entdeckt
9. **branch too far** — eine Verzweigung führt über eine zu große Distanz.
10. **label declared twice** — ein Labelname wurde zweimal benutzt.
11. **too many labels** — Label und Quelltext passen zusammen nicht mehr in den Speicher.
12. **no macro to close** — die Anzahl der .ma-Anweisungen stimmt nicht mit der Anzahl der .rt-Anweisungen überein.
13. **parameter** — im Makroaufruf stimmt die Parameterliste nicht mit der Parameterliste der Definition überein.
14. **return** — es liegt keine Rückkehradresse auf dem Stack, als eine .rt-Anweisung ausgeführt werden sollte.

Hinzuweisen ist noch auf eine einfache Möglichkeit, den »label twice-error« zu vermeiden:
Legt man eine Makrodefinition um einen beliebigen Block des Quelltextes, so sind alle Label in dem Block automatisch lokal. Auf diese Weise kann schon vorhandener Quelltext in neuen eingefügt werden, ohne daß man sich um doppelt verwendete Labelnamen kümmern muß.

Bild 2. Fehlermeldungen von »Hypra-Ass«

Nr.	Befehl	Beschreibung
1)	.ba $C000	gibt die Startadresse der Assemblierung an. Bei anderen Assemblern heißt dieser Befehl auch org oder * = .
2)	.eq label=wert	weist einem Label einen Wert zu
3)	.gl label=wert	weist einem globalen Label einen Wert zu
4)	.by 1,2,"a"	Einfügen von Byte-Werten in den Quelltext
5)	.wo 1234,label	Einfügen von Adressen in der Folge low/high
6)	.tx"text"	Einfügen von Text als ASCII-Werte
7)	.ap "file"	Verketten von Quelltexten
8)	.ob "file,p,w"	Senden des Objektcodes zur Floppy
9)	.en	Schließen des Objektfiles
10)	.on ausdruck,sprung	bedingter Sprung, wenn Ausdruck wahr
11)	.go sprung	unbedingter Sprung
12)	.if ausdruck	Fortführung der Assemblierung bei ELSE, falls Ausdruck falsch. Ansonsten hinter .if bis zu ELSE oder ENDIF.
13)	.el	Alternative zu den Zeilen, die hinter .if stehen
14)	.ei	Ende der IF-Konstruktion
15)	.co var1,var2	Übergabe von Labeln und Quelltext an nachgeladene Teile
16)	.ma makro (par1,par2)	Makrodefinitionszeile
17)	.rt	Ende der Makrodefinition
18)	...makro (par1,par2)	Makroaufruf
19)	.li, lfn, dn, ba	sendet formatiertes Listing unter der File-Nummer lfn zum Gerät dn mit der Sekundäradresse ba
20)	.sy lfn, dn, ba	sendet formatierte Symboltabelle unter der File-Nummer lfn zum Gerät dn mit der Sekundäradresse ba
21)	.st	beendet die Assemblierung
22)	.dp t0, t1, t2, t3	setzt die Tabulatoren T0, T1, T2, T3 aus dem Quelltext heraus

Vor den Anweisungen 12, 13, 14, 16 und 17 dürfen in derselben Zeile keine Label stehen.

Bild 3. Zusammenfassung aller Pseudobefehle

Adresse	Belegung
$0000	Zeropage
$033e	Bandpuffer als Zwischenspeicher
$0400	Video-RAM
$0800	Hypra-Ass
$1fd7	Raum für Quelltext und Label. Quelltext bis maximal $a000
$a000	Basic-Interpreter — darunter von c000 abwärts die Symboltabelle
$c000	frei!
$d000	I/O und so weiter
$c000	Kernal

Bild 4. Speicherbelegung von »Hypra-Ass«

```
programm : hypra-ass        0801 1fd8
-----------------------------------
0801 : 60 08 00 00 9e 32 39 30   26
0809 : 31 3a 22 0d 91 05 0e 27   49
0811 : 68 59 50 52 41 2d 61 53   2e
0819 : 53 20 20 6d 41 4b 52 4f   88
0821 : 2d 61 53 53 45 4d 42 4c   9e
0829 : 45 52 0d 0d 20 28 63 29   9f
0831 : 20 31 39 38 35 20 62 59   d0
0839 : 20 20 67 45 52 44 20 6d   8e
0841 : 4f 45 4c 4c 4d 41 4e 4e   84
0849 : 0d 0d 20 28 63 4f 52 52   89
0851 : 45 43 54 45 44 20 76 45   9f
0859 : 52 53 49 4f 4e 29 00 00   bf
0861 : 00 20 73 00 20 6b 08 4c   64
0869 : ae a7 f0 0c e9 80 90 03   93
0871 : 4c f3 a7 68 68 4c 08 af   16
0879 : 60 0b 43 50 58 43 50 59   cd
0881 : 4c 44 58 4c 44 59 43 4d   46
0889 : 50 41 44 43 41 4e 44 44   13
0891 : 45 43 45 4f 52 49 4e 43   e2
0899 : 4c 44 41 41 53 4c 42 49   b3
08a1 : 54 4c 53 52 4f 52 41 52   6c
08a9 : 4f 4c 52 4f 52 53 42 43   ec
08b1 : 53 54 41 53 54 58 53 54   e7
08b9 : 59 4a 4d 50 4a 53 52 54   c6
08c1 : 58 41 54 41 58 54 59 41   07
08c9 : 54 41 59 54 53 58 54 58   99
08d1 : 53 50 48 50 50 4c 50 50   b2
08d9 : 48 41 50 4c 41 42 52 4b   65
08e1 : 52 54 49 52 54 53 4e 4f   b2
08e9 : 50 43 4c 43 53 45 43 43   49
08f1 : 4c 49 53 45 49 43 4c 56   ec
08f9 : 43 4c 44 53 45 44 44 45   f0
0901 : 59 49 4e 59 44 45 58 49   20
0909 : 4e 58 42 50 4c 42 4d 49   bd
0911 : 42 56 43 42 56 53 42 43   27
0919 : 43 42 43 53 42 4e 45 42   e9
0921 : 45 51 e4 c4 a6 a4 c5 65   52
0929 : 25 c6 45 e6 a5 06 24 46   87
0931 : 05 26 66 e5 85 86 84 4c   d7
0939 : 20 8a aa 98 a8 ba 9a 08   37
0941 : 28 48 68 00 40 60 ea 18   8a
0949 : 38 58 78 b8 d8 f8 88 c8   eb
0951 : ca e8 10 30 50 70 90 b0   c6
0959 : d0 f0 40 40 54 68 7b 7b   27
0961 : 7b 28 7b 28 7b a8 00 a8   23
0969 : 7b a8 a8 7b 3b 04 08 00   c6
0971 : c0 00 c0 d8 1f 02 02 02   8b
0979 : 02 03 03 02 01 0c fc 10   83
0981 : 10 14 18 04 04 2b 2d 2a   c5
0989 : 2f 5e 41 4f 3e 3d 3c 43   67
0991 : 41 4e 27 54 20 4e 55 4d   b2
0999 : 42 45 52 20 54 45 52 4d   6a
09a1 : 00 4e 4f 20 4d 4e 45 4d   97
09a9 : 4f 4e 49 43 00 49 4c 4c   ee
09b1 : 45 47 41 4c 20 52 45 47   ac
09b9 : 49 53 54 45 52 00 45 4e   40
09c1 : 44 20 4f 46 20 4c 49 4e   d8
09c9 : 45 20 45 58 53 50 45 43   ce
09d1 : 54 45 44 00 57 52 4f 4e   bb
09d9 : 47 20 41 44 44 52 45 53   9c
09e1 : 53 00 42 52 41 4e 43 48   33
09e9 : 20 54 4f 4f 20 46 41 52   cf
09f1 : 00 55 4e 4b 4e 4f 57 4e   f2
09f9 : 20 4c 41 42 45 4c 00 49   21
0a01 : 4c 4c 45 47 41 4c 20 4c   3d
0a09 : 41 42 45 4c 00 54 4f 4f   c5
0a11 : 20 4d 41 4e 59 20 4c 41   3c
0a19 : 42 45 4c 53 00 4e 4f 20   6b
0a21 : 4d 41 43 52 4f 20 54 4f   10
0a29 : 20 43 4c 4f 53 45 00 55   f2
0a31 : 4e 4b 4e 4f 57 4e 20 50   ab
0a39 : 53 45 55 44 4f 00 4c 41   b5
0a41 : 42 45 4c 20 44 45 43 4c   51
0a49 : 41 52 45 44 20 54 57 49   22
0a51 : 43 45 00 50 41 52 41 4d   87
0a59 : 45 54 45 52 00 52 45 54   b4
0a61 : 55 52 4e 00 90 09 a2 09   61
0a69 : ae 09 bf 09 d5 09 e3 09   f4
0a71 : f2 09 00 0a 0e 0a 1e 0a   e7
0a79 : 30 0a 3f 0a 54 0a 0a aa   d3
0a81 : bd 65 0a 85 22 bd 66 0a   e2
0a89 : 85 23 a9 0e 20 c3 ff 20   2c
0a91 : cc ff a9 00 85 13 20 d7   e8
0a99 : aa a0 00 b1 22 f0 06 20   cc
0aa1 : d2 ff c8 d0 f6 4c 62 a4   64
0aa9 : a9 00 85 0d 20 73 00 b0   54
0ab1 : 06 20 f3 bc 4c 0c 0b 20   ed
0ab9 : 13 b1 90 06 20 b1 0d 4c   e6
0ac1 : 0c 0b 20 79 00 10 03 4c   af
0ac9 : b1 ae c9 2b f0 de c9 24   1e
0ad1 : f0 20 c9 2d f0 16 c9 22   14
0ad9 : f0 15 c9 21 f0 17 c9 3e   56
0ae1 : f0 5a c9 3c f0 56 20 f1   1e
0ae9 : ae 4c 0c 0b 4c 0d af 4c   a6
0af1 : bd ae 4c 6a 0d 20 73 00   05
0af9 : c9 4e d0 0a 20 73 00 c9   90
0b01 : 21 d0 03 4c d0 ae a9 00   fe
0b09 : 4c 7f 0a 20 79 00 aa 10   fe
0b11 : 01 60 68 68 8a c9 21 d0   87
0b19 : 0c 20 73 00 aa 20 73 00   8c
0b21 : c9 21 d0 e2 8a a2 0a dd   ad
0b29 : 85 09 f0 05 ca d0 f8 f0   09
0b31 : 04 8a 18 69 a9 a2 00 86   6a
0b39 : 4d 4c bb ad 48 20 73 00   a4
0b41 : 20 f1 ae 20 f7 b7 aa 68   c2
0b49 : c9 3c f0 02 8a a8 20 a2   60
0b51 : b3 4c 0c 0b a9 9d a2 17   cf
0b59 : 8d 02 03 8e 03 03 a9 a9   bc
0b61 : a2 0a 8d 0a 03 8e 0b 03   84
0b69 : a9 47 a2 0f 8d 08 03 8e   83
0b71 : 09 03 a9 1f 85 2c a9 d8   5c
0b79 : 85 2b a9 00 8d d7 1f 20   53
0b81 : 44 a6 a2 fa 9a 4c 49 a8   a3
0b89 : a2 00 86 41 a0 00 b1 7a   bb
0b91 : dd 7b 08 d0 10 c8 b1 7a   4b
0b99 : dd 7c 08 d0 08 c8 b1 7a   53
0ba1 : dd 7d 08 f0 0e e6 41 e8   4c
0ba9 : e8 e8 e0 a8 d0 de a9 01   ff
0bb1 : 4c 7f 0a a6 41 bd 23 09   b5
0bb9 : 85 3b a9 02 4c fc a8 a9   29
0bc1 : 00 85 3e 20 73 00 c9 23   bc
0bc9 : f0 4f c9 28 d0 55 20 2a   65
0bd1 : 14 a5 3d d0 14 20 79 00   49
0bd9 : c9 2c f0 22 c9 29 d0 19   94
0be1 : 20 73 00 c9 2c f0 27 d0   7d
0be9 : 03 20 73 00 a5 41 c9 15   8f
0bf1 : d0 07 a9 09 85 3e 4c 6d   27
0bf9 : 0c a9 04 4c 7f 0a 20 73   14
0c01 : 00 a9 58 20 ff ae 20 f7   d6
0c09 : ae e6 3e 4c 6d 0c 20 73   e2
0c11 : 00 a9 59 20 ff ae 4c 6d   c2
0c19 : 0c 20 38 14 a9 06 85 3e   23
0c21 : 4c 6d 0c a9 07 85 3e 20   32
0c29 : 79 00 f0 3b c9 3b f0 37   ee
0c31 : a2 05 86 3e 20 2d 14 20   bb
0c39 : 79 00 c9 2c d0 22 20 73   30
0c41 : 00 aa 20 73 00 8a c9 58   39
0c49 : f0 0b c9 59 f0 05 a9 02   3e
0c51 : 4c 7f 0a c6 3e a5 3d d0   60
0c59 : 04 c6 3e c6 3e 4c 6d 0c   3d
0c61 : a9 0a 85 3e 4c 56 0c a9   33
0c69 : 07 85 3e 60 20 79 00 f0   7e
0c71 : fa c9 3b f0 f6 a9 03 4c   9e
0c79 : 7f 0a a6 3e e0 06 d0 0c   08
0c81 : a5 41 c9 04 b0 06 a5 3b   02
0c89 : e9 07 85 3b e0 09 f0 29   2b
0c91 : a5 41 c9 15 90 35 f0 14   8a
0c99 : c9 16 f0 10 c9 30 [illegible] 03   12
0ca1 : 4c 28 0d e0 07 d0 1f a9   27
0ca9 : 01 85 42 60 e0 08 f0 04   24
0cb1 : e0 0a d0 12 a9 03 85 42   5a
0cb9 : 60 a5 41 c9 15 d0 07 a9   bd
0cc1 : 6c 85 3b 4c b5 0c a9 04   b3
0cc9 : 4c 7f 0a a9 02 85 42 e0   a4
0cd1 : 08 f0 0d e0 0a d0 0a e6   ce
0cd9 : 42 a5 3b 18 69 08 85 3b   23
0ce1 : 60 a9 01 e0 00 f0 05 0a   22
0ce9 : ca 4c e4 0c a6 41 3d 5b   b4
0cf1 : 09 d0 11 a6 3e e0 02 f0   50
0cf9 : 04 e0 03 d0 c9 e8 e8 86   dd
0d01 : 3e 4c e2 0c a6 3e e0 04   87
0d09 : d0 0f a5 41 c9 02 d0 09   f4
0d11 : a9 03 85 42 a9 be 85 3b   03
0d19 : 60 bd 76 09 85 42 bd 7e   75
0d21 : 09 18 65 3b 85 3b 60 20   eb
0d29 : ad 0c c6 42 a5 fd c9 02   4c
0d31 : 90 f4 a5 3c 38 e5 fb aa   24
0d39 : a5 3d e5 fc a8 8a e9 02   20
0d41 : 85 3c b0 01 88 c0 00 f0   a1
0d49 : 10 c8 d0 12 c9 80 90 0e   33
0d51 : e6 3c c9 fe 90 02 c6 3c   54
0d59 : 60 c9 80 b0 01 60 a9 05   98
0d61 : 4c 7f 0a ed c9 80 90 e9   64
0d69 : 60 a2 0a a9 00 95 5d ca   8a
0d71 : d0 fb 20 73 00 90 0b 20   a6
0d79 : 13 b1 90 17 e9 07 c9 40   ea
0d81 : b0 11 e9 2f 20 7e bd a5   52
0d89 : 61 18 69 04 85 61 90 e2   3d
0d91 : 4c 7e b9 a5 61 e9 03 90   d2
0d99 : 05 85 61 4c 0c 0b 4c 08   9d
0da1 : af 85 62 84 63 a2 90 38   3a
0da9 : 4c 49 bc 00 00 01 00 00   d1
0db1 : a2 00 86 0d 86 0c 86 0e   96
0db9 : a5 7a a4 7b 85 49 84 4a   7d
0dc1 : 20 79 00 20 13 b1 b0 09   36
0dc9 : a9 20 85 81 a9 07 4c 7f   17
0dd1 : 0a 20 73 00 90 05 20 13   a0
0dd9 : b1 90 04 e8 4c d2 0d c9   14
0de1 : 27 f0 f8 e8 86 45 a9 36   81
0de9 : 85 01 a9 f9 a2 bf 86 60   9b
0df1 : 85 5f e4 30 90 4a d0 04   0c
0df9 : c5 2f 90 44 a0 01 b1 5f   9a
0e01 : c9 ff f0 0e cd ad 0d d0   e8
0e09 : 2a 88 b1 5f cd ac 0d d0   e8
0e11 : 22 c8 c8 b1 5f c5 45 d0   da
0e19 : 1a c8 b1 5f 85 47 c8 b1   09
0e21 : 5f 85 48 a0 00 b1 49 d1   bf
0e29 : 47 d0 08 c8 c4 45 d0 f5   99
0e31 : 4c a5 0e a5 5f 38 e9 07   f6
0e39 : a6 60 b0 b4 ca 4c ef 0d   bb
0e41 : 68 48 c9 bf d0 19 20 9f   cd
0e49 : 0e a6 fd f0 08 ca f0 08   f2
0e51 : a9 06 4c 7f 0a a9 00 2c   47
0e59 : a9 80 a0 00 4c a2 0d a5   c4
0e61 : 5f a4 60 c4 2e 90 06 d0   e4
0e69 : 07 c5 2d b0 03 4c b7 0e   42
0e71 : 85 2f 84 30 a0 00 ad ac   cf
0e79 : 0d 91 5f c8 ad ad 0d 91   df
0e81 : 5f c8 a5 45 91 5f c8 a5   d9
0e89 : 49 91 5f c8 a5 4a 91 5f   3d
0e91 : a5 5f a4 60 18 69 05 90   1d
0e99 : 01 c8 85 49 84 4a a9 37   39
0ea1 : 85 01 38 60 20 91 0e a9   db
0ea9 : 36 85 01 20 83 14 20 9f   7f
0eb1 : 0e 20 a6 0d 18 60 20 9f   5f
0eb9 : 0e a9 08 4c 7f 0a 00 00   70
0ec1 : db 23 a9 00 85 fd 85 fe   f5
0ec9 : 4c 62 08 a0 02 b1 7a f0   d6
0ed1 : 1f c8 b1 7a 85 39 c8 b1   b9
0ed9 : 7a 85 3a a4 7b a6 7a e8   e2
0ee1 : d0 01 c8 8e c1 0e 8c c2   7a
0ee9 : 0e a0 04 20 fb a8 18 60   73
0ef1 : 38 60 00 0f 1e 02 0f a5   b5
0ef9 : fd c9 02 90 43 ad 32 16   04
0f01 : 20 c3 ff a9 ff 85 3a a9   a0
0f09 : 00 8d ac 0d 8d ad 0d 8d   32
0f11 : af 0d a9 01 8d ae 0d ad   af
0f19 : e0 16 f0 12 20 d7 aa 20   2e
0f21 : 03 17 20 d7 aa 20 d7 aa   13
0f29 : ae db 16 20 c9 ff 20 3e   e8
0f31 : 15 20 a6 15 20 cc ff ad   66
0f39 : db 16 20 c3 ff 4c 7b e3   b8
0f41 : 20 78 14 4c e9 0f 24 9d   0f
0f49 : 10 03 4c c3 0e 20 1f 15   ef
0f51 : a2 fa 9a 20 8e a6 a5 2b   26
0f59 : a6 2c 8d 74 09 8e 75 09   f4
0f61 : a9 00 48 a2 c0 8d e0 16   99
0f69 : 8d 33 16 8d 72 09 8e 73   58
0f71 : 09 ad 72 09 ae 73 09 85   c4
0f79 : 2f 86 30 a9 00 8d c0 0e   b8
0f81 : 8d bf 0e 8d ac 0d 8d ad   e8
0f89 : 0d 8d af 0d a9 01 8d ae   21
0f91 : 0d 20 cc 0e b0 52 20 73   a8
0f99 : 00 c9 2e d0 42 20 73 00   16
0fa1 : c9 4d f0 1e c9 52 d0 37   f2
0fa9 : 20 73 00 c9 54 d0 10 ee   a6
0fb1 : c0 0e ad bf 0e cd c0 0e   4a
0fb9 : b0 25 a9 09 4c 7f 0a 4c   09
0fc1 : 08 af 20 73 00 a9 41 20   aa
0fc9 : ff ae 20 b1 0d a0 00 a5   7f
0fd1 : 7a 91 49 c8 a5 7b 91 49   8e
0fd9 : a9 ff 91 5f ee bf 0e 20   37
0fe1 : 09 a9 20 fb a8 4c 92 0f   9c
0fe9 : e6 fd a9 00 8d ac 0d 8d   c6
0ff1 : ad 0d 8d af 0d a9 01 8d   bb
0ff9 : ae 0d 20 8e a6 20 b4 16   72
1001 : 20 cc 0e 90 06 4c f8 0e   e0
1009 : 4c 96 10 a0 01 b1 7a c9   d4
1011 : 3b f0 f5 c9 20 f0 4e c9   d1
1019 : 2e f0 4a a5 fd c9 01 f0   1b
1021 : 0f c8 b1 7a f0 5b c9 20   a1
1029 : d0 f7 20 fb a8 4c 69 10   2f
1031 : a9 ff 85 81 20 73 00 20   49
1039 : b1 0d a9 20 85 81 b0 15   31
1041 : a5 4a cd 73 09 90 09 d0   c8
1049 : 0c a5 49 cd 72 09 b0 05   70
1051 : a9 0b 4c 7f 0a a5 fc a6   92
1059 : fb 20 91 14 20 79 00 c9   ad
1061 : 00 d0 08 f0 1c 20 73 00   7a
1069 : 20 79 00 c9 2e f0 55 20   7f
1071 : 5f 14 a0 00 b1 7a f0 14   dd
1079 : c9 3b f0 10 a9 03 4c 7f   01
1081 : 0a ad 33 16 f0 03 20 65   64
1089 : 17 20 73 00 20 09 a9 20   be
1091 : fb a8 4c fe 0f ad 33 16   2b
1099 : f0 ef ae 32 16 20 c9 ff   fc
10a1 : 20 57 17 ad c1 0e ae c2   b5
10a9 : 0e 85 5f 86 60 a2 00 20   7e
10b1 : 82 1b 20 d7 aa 20 cc ff   a3
10b9 : 20 73 00 20 09 a9 20 fb   ed
10c1 : a8 4c 01 10 a2 00 a0 01   80
10c9 : b1 7a dd 9c 14 d0 08 c8   3c
10d1 : b1 7a dd 9d 14 f0 0b e8   b1
10d9 : e8 e0 2c d0 e9 a9 0a 4c   03
10e1 : 7f 0a bd c9 14 48 bd c8   1a
10e9 : 14 48 ad 33 16 f0 03 20   28
10f1 : 65 17 a9 02 4c fc a8 20   1c
10f9 : 73 00 20 b1 0d a9 ff a0   0a
1101 : 01 91 5f 4c 0d 11 20 73   ed
1109 : 00 20 b1 0d a9 3d 20 ff   2c
1111 : ae a5 4a 48 a5 49 48 20   33
1119 : 2d 14 68 85 49 68 85 4a   9e
1121 : a5 3d a6 3c 20 91 14 4c   0d
1129 : 73 10 20 2a 14 8d 71 09   77
```

Listing zum Programm »Hypra-Ass«. Bitte beachten Sie die Eingabehinweise auf Seite 110



```
1131 : 8c 70 09 20 78 14 4c 73  7c
1139 : 10 a9 01 85 42 20 38 14  3d
1141 : 85 3b 20 d3 13 20 79 00  fe
1149 : c9 2c f0 f1 4c 73 10 a9  97
1151 : 01 85 42 a0 00 20 73 00  88
1159 : c9 22 f0 03 4c 99 ad c8  aa
1161 : b1 7a f0 11 c9 22 f0 0c  37
1169 : 85 3b 98 48 20 d3 13 68  79
1171 : a8 4c 60 11 c8 20 fb a8  48
1179 : 4c 73 10 a9 02 85 42 20  4e
1181 : 2a 14 85 3c 84 3b 20 d3  e8
1189 : 13 20 79 00 c9 2c f0 ef  ac
1191 : 4c 73 10 a6 fd ca f0 2a  be
1199 : 20 e7 ff 20 73 00 20 f4  52
11a1 : 14 a9 0e a0 6e a2 08 20  7e
11a9 : ba ff 20 c0 ff a2 0e 20  10
11b1 : c9 ff a5 fb 20 d2 ff a5  46
11b9 : fc 20 d2 ff 20 cc ff 4c  7b
11c1 : 8d 10 a9 80 85 fe 4c 8d  6d
11c9 : 10 a5 fd c9 02 90 0c a9  8d
11d1 : 00 85 fe a9 0e 20 c3 ff  7a
11d9 : 20 cc ff 4c 8d 10 20 09  d5
11e1 : a9 20 fb a8 a0 04 20 fb  51
11e9 : a8 20 73 00 c9 2e d0 ee  ad
11f1 : 20 73 00 c9 52 d0 e7 4c  e8
11f9 : 8d 10 ad ac 0d 0d ad 0d  99
1201 : 8d b0 0d 20 73 00 20 b1  49
1209 : 0d a5 7b 48 8d d2 13 a5  d8
1211 : 7a 48 8d d1 13 a9 36 48  35
1219 : 85 01 a0 01 b1 49 85 7b  da
1221 : 88 b1 49 85 7a a9 37 85  62
1229 : 01 20 79 00 c9 28 d0 4e  56
1231 : 20 bc 13 20 79 00 c9 28  87
1239 : d0 4a 20 2a 14 20 bc 13  d7
1241 : 20 8a 12 20 73 00 20 b1  4a
1249 : 0d a5 3d a6 3c 20 91 14  80
1251 : 20 79 00 c9 29 f0 14 c9  65
1259 : 2c d0 29 20 bc 13 20 79  14
1261 : 00 c9 2c d0 1f 20 ae 12  3d
1269 : 4c 3b 12 20 bc 13 20 79  b3
1271 : 00 c9 29 d0 0f 20 bc 13  c5
1279 : 20 a5 12 4c 8d 10 20 8a  69
1281 : 12 4c 79 12 a9 0c 4c 7f  85
1289 : 0a ad b0 0d d0 0d ad ae  c1
1291 : 0d ae af 0d 8d ac 0d 8e  12
1299 : ad 0d 60 ee ac 0d d0 03  3f
12a1 : ee ad 0d 60 ee ae 0d d0  ef
12a9 : 03 ee af 0d 60 ad b0 0d  01
12b1 : d0 09 a9 00 8d ac 0d 8d  fe
12b9 : ad 0d 60 ad ac 0d d0 03  37,
12c1 : ce ad 0d ce ac 0d 60 68  08
12c9 : c9 36 d0 1d 68 85 7a 68  f3
12d1 : 85 7b 68 48 c9 36 d0 06  d5
12d9 : 20 bc 12 4c 8d 10 a9 00  65
12e1 : 8d ac 0d 8d ad 0d 4c 8d  49
12e9 : 10 a9 0d 4c 7f 0a a6 fd  7a
12f1 : ca d0 36 20 73 00 f0 14  d8
12f9 : c9 3b f0 10 20 b1 0d 20  a2
1301 : 79 00 c9 2c d0 06 20 73  17
1309 : 00 4c fd 12 a5 2f a6 30  c0
1311 : 8d 72 09 8e 73 09 20 09  fe
1319 : a9 c8 a6 7b 18 98 65 7a  10
1321 : 90 01 e8 8d 74 09 8e 75  d2
1329 : 09 4c 8d 10 a5 fd c9 02  33
1331 : 90 f7 20 cc ff a5 fb a6  c9
1339 : fc 8d 70 09 8e 71 09 20  12
1341 : 73 00 20 f4 14 a2 08 a0  13
1349 : 00 20 ba ff a9 00 85 0a  cd
1351 : ae 74 09 ac 75 09 20 75  1c
1359 : e1 a9 00 85 fd a5 fe f0  aa
1361 : 05 a2 0e 20 c9 ff 4c 72  f2
1369 : 0f 20 73 00 20 8a ad a5  be
1371 : 61 d0 0c 20 fd ae 20 2d  72
1379 : 14 20 a3 a8 4c fe 0f 4c  2d
1381 : 8d 10 20 2a 14 4c 7a 13  17
1389 : 20 73 00 20 8a ad a5 61  d6
1391 : f0 03 4c 73 10 20 09 a9  fe
1399 : c8 c8 c8 c8 20 fb a8 20  d5
13a1 : 73 00 c9 2e d0 ef 20 73  40
13a9 : 00 c9 45 d0 e8 20 73 00  57
13b1 : c9 4c f0 04 c9 49 d0 dd  43
13b9 : 4c 8d 10 a6 7a a4 7b ad  bb
13c1 : d1 13 85 7a ad d2 13 85  95
13c9 : 7b 8e d1 13 8c d2 13 60  ce
13d1 : 00 00 a6 fd ca f0 24 a5  4b
13d9 : fe d0 2c a5 fb a6 fc 85  f3
13e1 : 22 86 23 a6 42 a0 00 a5  58
13e9 : 3b 91 22 ca f0 0d c8 a5  b5
13f1 : 3c 91 22 ca f0 05 c8 a5  7d
13f9 : 3d 91 22 a5 42 18 65 fb  af
1401 : 85 fb 90 02 e6 fc 60 a2  05
1409 : 0e 20 c9 ff a6 42 a5 3b  23
1411 : 20 d2 ff ca f0 0d a5 3c  7a
1419 : 20 d2 ff ca f0 05 a5 3d  44
1421 : 4c 19 14 20 cc ff 4c fc  fb
1429 : 13 20 73 00 20 8a ad 20  77
1431 : f7 b7 85 3d 84 3c 60 20  f9
1439 : 73 00 c9 22 f0 09 20 2d  95
1441 : 14 c9 00 d0 16 98 60 a0  3d
1449 : 01 b1 7a 85 3c c8 b1 7a  38
1451 : c9 22 d0 07 c8 20 fb a8  0f
1459 : a5 3c 60 4c 48 b2 20 89  6c
1461 : 0b 20 c0 0b 20 7b 0c a6  69
1469 : fd ca f0 08 ad 33 16 f0  b7
1471 : 03 20 4e 16 4c d3 13 ad  e6
1479 : 70 09 ae 71 09 85 fb 86  01
1481 : fc 60 a0 01 b1 49 85 62  36
1489 : 88 b1 49 85 63 84 70 60  ca
1491 : a0 01 91 49 88 8a 91 49  f5
1499 : 84 70 60 45 51 42 41 42  c7
14a1 : 59 54 58 57 4f 4f 42 45  28
14a9 : 4e 4d 41 2e 2e 52 54 43  01
14b1 : 4f 41 50 4f 4e 47 4f 49  8e
14b9 : 46 45 4c 45 49 47 4c 4c  f6
14c1 : 49 53 59 53 54 44 50 06  29
14c9 : 11 2a 11 39 11 4f 11 7b  22
14d1 : 11 93 11 c9 11 de 11 fa  6c
14d9 : 11 c7 12 ee 12 2c 13 69  d2
14e1 : 13 82 13 88 13 95 13 b8  a7
14e9 : 13 f7 10 f2 15 bc 16 3c  62
14f1 : 17 78 17 a6 7a a4 7b e8  6b
14f9 : d0 01 c8 86 bb 84 bc a0  61
1501 : 00 c8 b1 7a c9 22 d0 f9  06
1509 : 88 84 b7 60 45 4e 44 20  e5
1511 : 4f 46 20 41 53 53 45 4d  33
1519 : 42 4c 59 20 20 00 ad 0e  b1
1521 : dc 09 80 8d 0e dc ad 0f  f0
1529 : dc 29 fe 8d 0f dc a9 00  8a
1531 : 8d 0b dc 8d 0a dc 8d 09  fc
1539 : dc 8d 08 dc 60 a9 00 8d  e8
1541 : 0b dc 20 d7 aa 20 57 17  f5
1549 : a9 0d a0 15 20 1e ab ad  41
1551 : 0a dc 29 0f 18 69 30 20  c3
1559 : d2 ff a9 3a 20 d2 ff ad  d0
1561 : 09 dc aa 29 f0 4a 4a 4a  c7
1569 : 4a 18 69 30 20 d2 ff 8a  cd
1571 : 29 0f 18 69 30 20 d2 ff  a4
1579 : a9 2e 20 d2 ff ad 08 dc  e3
1581 : 29 0f 18 69 30 20 d2 ff  b4
1589 : 4c d7 aa 42 41 53 45 20  b8
1591 : 3d 20 24 00 20 20 4c 41  9e
1599 : 53 54 20 42 59 54 45 20  f4
15a1 : 41 54 20 24 00 20 57 17  26
15a9 : a9 8c a0 15 20 1e ab ad  60
15b1 : 71 09 20 37 16 ad 70 09  38
15b9 : 20 37 16 a9 95 a0 15 20  23
15c1 : 1e ab a4 fc a6 fb d0 01  0d
15c9 : 88 ca 98 20 37 16 8a 20  6f
15d1 : 37 16 4c d7 aa 48 59 50  14
15d9 : 52 41 2d 41 53 53 20 20  d0
15e1 : 41 53 53 45 4d 42 4c 45  ec
15e9 : 52 4c 49 53 54 49 4e 47  75
15f1 : 3a 00 a6 fd ca f0 37 20  e6
15f9 : 73 00 20 19 e2 a5 b8 8d  f1
1601 : 32 16 20 c0 ff b0 2c a9  e8
1609 : ff 8d 33 16 ae 32 16 20  74
1611 : c9 ff 20 d7 aa 20 d7 aa  3d
1619 : 20 57 17 a9 d6 a0 15 20  e7
1621 : 1e ab 20 d7 aa 20 d7 aa  78
1629 : 20 cc ff 20 65 17 4c 8d  0f
1631 : 10 00 00 4c f9 e0 48 4a  27
1639 : 4a 4a 4a 20 42 16 68 29  08
1641 : 0f 18 69 30 c9 3a 90 02  71
1649 : 69 06 4c d2 ff ae 32 16  8d
1651 : 20 c9 ff a2 00 ec f6 0e  09
1659 : b0 08 a9 20 20 d2 ff e8  e6
1661 : d0 f3 a5 fc 20 37 16 a5  93
1669 : fb 20 37 16 20 52 17 a6  43
1671 : 42 a5 3b 20 37 16 ca f0  8a
1679 : 10 a5 3c 20 37 16 ca f0  a0
1681 : 0b a5 3d 20 37 16 4c 90  29
1689 : 16 20 4d 17 20 4d 17 20  ef
1691 : 52 17 a9 3a 20 d2 ff 4c  52
1699 : cc ff ad c1 0e ae c2 0e  86
16a1 : 85 5f 86 60 ae 32 16 20  99
16a9 : c9 ff 20 4d 1b 20 d7 aa  8b
16b1 : 4c cc ff ad 33 16 f0 03  c7
16b9 : 4c 9b 16 60 a6 fd ca f0  cc
16c1 : 15 20 73 00 20 19 e2 a2  5f
16c9 : 05 b5 b7 9d da 16 ca 10  f4
16d1 : f8 a9 ff 8d e0 16 4c 8d  5a
16d9 : 10 00 00 00 00 00 00 00  ea
16e1 : 4c f9 e0 53 59 4d 42 4f  74
16e9 : 4c 53 20 49 4e 20 41 4c  94
16f1 : 50 48 41 42 45 54 49 43  a1
16f9 : 41 4c 20 4f 52 44 45 52  53
1701 : 3a 00 a2 05 bd da 16 95  bb
1709 : b7 ca 10 f8 20 c0 ff b0  b2
1711 : cf ae db 16 20 c9 ff 20  82
1719 : d7 aa 20 d7 aa 20 d7 aa  a9
1721 : 20 57 17 a9 e4 a0 16 20  d4
1729 : 1e ab 20 d7 aa 20 d7 aa  80
1731 : 20 7a 1e 20 90 1d 20 d7  3c
1739 : aa 4c cc ff a6 fd ca f0  a4
1741 : 08 a9 0e 20 c3 ff 4c f8  05
1749 : 0e 4c 8d 10 a9 20 20 d2  a4
1751 : ff a9 20 4c d2 ff a2 00  6e
1759 : ec f6 0e b0 06 20 52 17  33
1761 : e8 d0 f5 60 ae 32 16 20  50
1769 : c9 ff 20 57 17 a2 0d 20  20
1771 : 52 17 ca d0 fa 4c cc ff  61
1779 : 20 38 14 8d f4 0e 20 fd  a8
1781 : ae 20 3f 14 8d f5 0e 20  93
1789 : fd ae 20 3f 14 8d f6 0e  73
1791 : 20 fd ae 20 3f 14 8d f7  1a
1799 : 0e 4c 8d 10 a9 00 85 fd  df
17a1 : 85 fc a6 02 f0 0c 30 42  43
17a9 : ca ca f0 2d ca f0 2d ca  39
17b1 : f0 30 20 60 a5 86 7a 84  4f
17b9 : 7b 20 73 00 aa f0 f3 c9  b7
17c1 : 5f f0 13 c9 2f f0 18 a2  b6
17c9 : ff 86 3a 20 79 00 90 2c  d0
17d1 : 20 79 a5 4c e1 a7 4c 1e  69
17d9 : 1f 4c 9c 1a 4c 81 18 4c  53
17e1 : 5c 1e 4c 48 1c a9 00 85  83
17e9 : 02 60 a5 02 29 7f 85 02  6e
17f1 : 20 6b 18 20 4d 1b 20 d7  af
17f9 : aa 4c 9d 17 20 6b a9 a6  65
1801 : 7a e8 a0 05 bd 00 02 99  cf
1809 : fb 01 f0 49 c9 3b f0 27  73
1811 : c9 2e f0 23 c9 20 f0 05  fd
1819 : e8 c8 4c 05 18 e8 bd 00  d9
1821 : 02 f0 31 c9 20 f0 f6 c9  1a
1829 : 22 f0 18 c9 3b f0 07 c8  eb
1831 : 99 fb 01 4c 1e 18 c8 99  8b
1839 : fb 01 e8 bd 00 02 d0 f6  e8
1841 : 4c 55 18 c8 99 fb 01 e8  a6
1849 : bd 00 02 f0 07 c9 22 d0  8e
1851 : f2 4c 30 18 c8 99 fd 01  cc
1859 : 99 fb 01 c6 7b a9 ff 85  19
1861 : 7a a5 02 09 80 85 02 4c  24
1869 : a2 a4 a9 91 20 d2 ff a2  d8
1871 : 00 a9 20 20 d2 ff e8 e0  e4
1879 : 27 d0 f6 a9 00 85 d3 60  37
1881 : ad 3d 03 ae 3c 03 20 cd  5b
1889 : bd a2 04 20 2a 1f a9 2d  39
1891 : 20 d2 ff ad 3d 03 85 15  fc
1899 : ad 3c 03 85 14 18 6d 40  0e
18a1 : 03 8d 3c 03 90 03 ee 3d  32
18a9 : 03 20 60 a5 86 7a c8 84  f1
18b1 : 7b a2 00 ad 00 02 d0 06  92
18b9 : 20 e6 17 4c 9d 17 4c 03  65
18c1 : 18 e6 7a 20 eb b7 a5 14  2a
18c9 : a4 15 8d 3c 03 8c 3d 03  72
18d1 : 8e 40 03 a9 03 85 02 4c  72
18d9 : 9d 17 20 73 00 90 09 f0  03
18e1 : 07 c9 2d f0 03 4c 08 af  48
18e9 : 20 6b a9 20 13 a6 20 79  07
18f1 : 00 f0 0c c9 2d d0 ee 20  fb
18f9 : 73 00 20 6b a9 d0 e6 a5  ea
1901 : 14 05 15 d0 06 a9 ff 85  b0
1909 : 14 85 15 60 a0 01 b1 5f  c9
1911 : f0 13 c8 b1 5f aa c8 b1  c5
1919 : 5f c5 15 d0 04 e4 14 f0  54
1921 : 02 b0 03 18 24 38 60 a0  06
1929 : 00 b1 5f aa c8 b1 5f 85  d2
1931 : 60 86 5f 60 4c 48 b2 20  ca
1939 : db 18 a5 5f a6 60 85 19  2b
1941 : 86 1a a5 14 25 15 aa e8  38
1949 : f0 06 e6 14 d0 02 e6 15  5b
1951 : 20 13 a6 a5 5f a6 60 85  11
1959 : 24 86 25 38 e5 19 8a e5  2e
1961 : 1a 90 d1 a5 2d e5 24 85  8a
1969 : 5f a5 2e e5 25 85 60 18  13
1971 : a5 19 65 5f 85 2d a5 1a  75
1979 : 65 60 85 2e a0 00 b1 24  4f
1981 : 91 19 e6 19 d0 02 e6 1a  68
1989 : e6 24 d0 02 e6 25 a5 5f  e3
1991 : 38 e9 01 85 5f a5 60 e9  27
1999 : 00 85 60 10 e1 20 59 a6  48
19a1 : 20 33 a5 4c 74 a4 20 db  f2
19a9 : 18 20 2c a8 20 0d 19 b0  22
19b1 : 0c 20 4d 1b 20 d7 aa 20  30
19b9 : 28 19 4c aa 19 4c 74 a4  e5
19c1 : 20 9b b7 86 fe 20 fd ae  b4
19c9 : 20 8a ad 20 f7 b7 a6 fe  73
19d1 : 9d 61 03 98 9d 43 03 20  33
19d9 : fd ae 20 8a ad 20 f7 b7  b2
19e1 : a6 fe 9d 9d 03 98 9d 7f  8c
19e9 : 03 4c 74 a4 bd 43 03 85  d1
19f1 : 14 bd 61 03 85 15 86 fe  b6
19f9 : 20 13 a6 a6 fe bd 7f 03  03
1a01 : 85 14 bd 9d 03 85 15 60  25
1a09 : 20 9e b7 20 ed 19 20 44  1b
1a11 : e5 4c aa 19 20 9b b7 86  b5
1a19 : fe 20 fd ae 20 eb b7 8e  da
1a21 : 40 03 a5 14 a6 15 8d 3c  90
1a29 : 03 8e 3d 03 a6 fe 20 ed  e2
1a31 : 19 ad 3c 03 9d 43 03 ad  ec
1a39 : 3d 03 9d 61 03 20 0d 19  23
1a41 : b0 1d ad 3d 03 91 5f 88  de
1a49 : ad 3c 03 91 5f 18 6d 40  f4
1a51 : 03 8d 3c 03 90 03 ee 3d  e2
1a59 : 03 20 28 19 4c 3e 1a ad  14
1a61 : 3c 03 38 ed 40 03 a6 fe  9f
1a69 : 9d 7f 03 b0 03 ce 3d 03  3e
1a71 : ad 3d 03 9d 9d 03 4c 74  3d
```

Listing zum Programm »Hypra-Ass«. Bitte beachten Sie die Eingabehinweise auf Seite 110 (Fortsetzung)

```
1a79 : a4 a9 02 85 02 a6 2c a5  74
1a81 : 2b 8e 3d 03 8d 3c 03 a9  bd
1a89 : ff 85 14 85 15 20 13 a6  ed
1a91 : a5 5f a6 60 85 2b 86 2c  c0
1a99 : 4c c6 1a ae 3d 03 ad 3c  c0
1aa1 : 03 86 2c 85 2b 20 e6 17  21
1aa9 : 4c 74 a4 20 9b b7 8e 49  a1
1ab1 : 1f 4c 9d 17 20 73 00 20  1e
1ab9 : 57 e2 a0 00 ae 49 1f 4c  f4
1ac1 : ba ff a9 01 2c a9 00 85  20
1ac9 : 0a 20 b5 1a 20 6f e1 4c  32
1ad1 : 74 a4 20 b5 1a 20 59 e1  22
1ad9 : 4c 74 a4 20 9b b7 20 ed  60
1ae1 : 19 20 83 1c 20 fd ae 20  5c
1ae9 : ad 1c 20 0d 19 90 06 20  bd
1af1 : e6 17 4c 74 a4 a6 60 a5  51
1af9 : 5f 18 69 04 85 5d 90 01  c7
1b01 : e8 86 5e a9 03 85 45 a0  ac
1b09 : ff e6 45 c8 b1 5d f0 22  f4
1b11 : c4 ba f0 22 d1 bb f0 f3  59
1b19 : b1 bb c9 3f f0 ed c8 b1  07
1b21 : 5d d0 fb c4 ba f0 1e 90  4a
1b29 : 1c e6 5d d0 da e6 5e 4c  21
1b31 : 08 1b c4 ba d0 0f a5 02  6f
1b39 : c9 04 d0 03 4c 04 1c 20  2e
1b41 : 4d 1b 20 d7 aa 20 28 19  9d
1b49 : 4c eb 1a 03 a0 02 b1 5f  11
1b51 : aa c8 b1 5f 85 62 86 63  04
1b59 : a2 90 38 20 49 bc 20 df  10
1b61 : bd 20 87 b4 20 a6 b6 8d  d4
1b69 : 4c 1b 20 24 ab ae 4c 1b  67
1b71 : e0 05 b0 08 a9 20 20 d2  c3
1b79 : ff e8 d0 f4 a9 2d 20 d2  e9
1b81 : ff a0 04 b1 5f c9 3b f0  1b
1b89 : 57 c9 2e f0 53 b1 5f f0  91
1b91 : 5b c9 20 f0 08 20 d2 ff  c4
1b99 : e8 c8 4c 8e 1b 20 d2 ff  c8
1ba1 : c8 e8 ec f4 0e b0 08 a9  91
1ba9 : 20 20 d2 ff 4c a2 1b a9  27
1bb1 : 03 85 22 b1 5f f0 35 20  c8
1bb9 : d2 ff c8 e8 c6 22 d0 f3  83
1bc1 : 20 3f ab e8 b1 5f f0 24  ab
1bc9 : c9 3b f0 07 20 d2 ff e8  b7
1bd1 : c8 d0 f1 ec f5 0e b0 08  be
1bd9 : a9 20 20 d2 ff e8 d0 f3  67
1be1 : b1 5f f0 08 20 d2 ff e8  e9
1be9 : c8 4c e1 1b 60 20 9b b7  98
1bf1 : 20 ed 19 20 83 1c 20 fd  e8
1bf9 : ae 20 95 1c a9 04 85 02  75
1c01 : 4c e5 1a a5 14 48 a5 15  bf
1c09 : 48 a2 05 a0 02 b1 5f 85  2e
1c11 : 14 c8 b1 5f 85 15 c8 c4  8f
1c19 : 45 f0 09 b1 5f 9d fb 01  24
1c21 : e8 4c 17 1c a0 00 c4 b7  05
1c29 : f0 0a b1 b8 9d fb 01 e8  31
1c31 : c8 4c 27 1c a4 ba b1 5d  0e
1c39 : 9d fb 01 f0 05 e8 c8 4c  86
1c41 : 37 1c 8a a8 4c a2 a4 20  eb
1c49 : 13 a6 20 4d 1b 20 d7 aa  c9
1c51 : 68 85 15 68 85 14 a6 fe  60
1c59 : bd 7f 03 85 14 bd 9d 03  f3
1c61 : 85 15 a5 45 18 65 b7 a8  60
1c69 : b1 5f f0 13 a6 60 98 18  68
1c71 : 65 5f 85 5d 90 01 e8 86  55
1c79 : 5e 88 84 45 4c 08 1b 4c  ef
1c81 : 46 1b a9 03 85 b9 85 bc  d5
1c89 : a9 b0 85 b8 a9 d8 85 bb  f2
1c91 : 60 4c 48 b2 20 9e ad 20  6e
1c99 : 82 b7 c9 26 b0 f3 85 b7  5e
1ca1 : a0 00 b1 22 91 b8 c8 c4  7d
1ca9 : b7 d0 f7 60 20 9e ad 20  c0
1cb1 : 82 b7 f0 dd c9 26 b0 d9  4b
1cb9 : 85 ba a0 00 b1 22 91 bb  ad
1cc1 : c8 c4 ba d0 f7 60 a9 08  ed
1cc9 : 85 ba 20 b4 ff a9 6f 85  60
1cd1 : b9 20 96 ff 20 a5 ff 20  af
1cd9 : d2 ff c9 0d d0 f6 20 ab  5b
1ce1 : ff 4c 74 a4 a9 24 8d 00  aa
1ce9 : 01 20 d7 aa a9 01 a8 a2  d0
1cf1 : 00 20 bd ff a2 08 a0 60  1e
1cf9 : 20 ba ff 20 d5 f3 a5 ba  83
1d01 : 20 b4 ff a5 b9 20 96 ff  27
1d09 : a9 00 85 90 a0 03 84 fb  52
1d11 : 20 a5 ff 85 fe a4 90 d0  ad
1d19 : 30 20 a5 ff a4 90 d0 29  27
1d21 : a4 fb 88 d0 e9 a6 fe 20  0f
1d29 : cd bd a9 20 20 d2 ff 20  1c
1d31 : a5 ff a6 90 d0 13 aa f0  c4
1d39 : 06 20 d2 ff 4c 30 1d a9  12
1d41 : 0d 20 d2 ff a0 02 4c 0f  7c
1d49 : 1d 20 42 f6 20 e6 17 4c  14
1d51 : c7 1c a9 08 a0 01 85 ba  2f
1d59 : 20 b1 ff a9 6f 85 b9 20  d1
1d61 : 93 ff b1 7a f0 07 20 a8  c9
1d69 : ff c8 4c 63 1d 20 ae ff  d9
1d71 : 4c 74 a4 a5 30 c5 2e d0  61
1d79 : 06 a5 2f c5 2d f0 0d 20  a5
1d81 : 73 00 c9 21 d0 03 20 7a  25
1d89 : 1e 20 90 1d 4c 74 a4 a2  bf
1d91 : bf a9 f9 85 45 86 46 e4  bf
1d99 : 30 90 7d d0 04 c5 2f 90  d7
1da1 : 77 20 2c a8 a9 36 85 01  ad
1da9 : a0 00 b1 45 c8 11 45 d0  2a
1db1 : 56 a0 06 b1 45 85 62 88  2a
1db9 : b1 45 85 63 88 b1 45 85  11
1dc1 : 23 88 b1 45 85 22 88 b1  2c
1dc9 : 45 85 47 a9 37 85 01 20  bc
1dd1 : d7 aa 20 57 17 a0 00 a2  ac
1dd9 : 00 b1 22 20 d2 ff [illegible] e8  61
1de1 : c4 47 d0 f5 e8 20 3f ab  1f
1de9 : ec f7 0e 90 f7 20 3f ab  3b
1df1 : a9 3d 20 d2 ff 20 3f ab  f1
1df9 : a9 24 20 d2 ff a5 62 20  0e
1e01 : 37 16 a5 63 20 37 16 a5  78
1e09 : 45 a6 46 38 e9 07 b0 01  d6
1e11 : ca a0 37 84 01 4c 94 1d  89
1e19 : 60 4d 56 4c 53 41 44 4e  2c
1e21 : 45 47 54 50 46 58 52 49  2c
1e29 : 4b 40 21 4f 42 55 43 1a  d7
1e31 : 1a 1a 1a 18 19 1a 19 1a  dd
1e39 : 1f 19 1a fc 1b 1c 1c 1d  48
1e41 : 1d 1f 1f 1f 1f 79 c2 c5  ee
1e49 : d2 c1 37 14 a6 ab 92 c0  e0
1e51 : db e1 ed e4 c6 52 73 2e  5e
1e59 : 49 81 b9 20 73 00 b0 03  d5
1e61 : 4c 09 1a a2 17 dd 19 1e  0e
1e69 : f0 06 ca d0 f8 4c 08 af  9a
1e71 : bd 2f 1e 48 bd 45 1e 48  66
1e79 : 60 a9 36 85 01 a5 30 48  7b
1e81 : a5 2f 48 20 e6 17 4c a4  76
1e89 : 1e a5 02 f0 0b a5 19 a6  a8
1e91 : 1a 85 2f 86 30 4c 84 1e  be
1e99 : a9 37 85 01 68 85 2f 68  a0
1ea1 : 85 30 60 a9 f9 a2 bf 85  4a
1ea9 : 19 86 1a 38 e9 07 85 1b  b6
1eb1 : b0 01 ca 86 1c e4 30 d0  b1
1eb9 : 06 90 ce c5 2f 90 ca a0  57
1ec1 : 02 b1 19 85 21 b1 1b 85  aa
1ec9 : 22 c8 b1 19 85 1d b1 1b  1d
1ed1 : 85 1f c8 b1 19 85 1e b1  e8
1ed9 : 1b 85 20 a0 00 b1 1d d1  79
1ee1 : 1f f0 04 b0 1a 90 11 c8  8b
1ee9 : c4 21 f0 04 c4 22 d0 ed  77
1ef1 : a5 21 c5 22 f0 02 b0 07  cc
1ef9 : a5 1b a6 1c 4c a8 1e a0  1d
1f01 : 06 b1 19 48 88 10 fa a0  65
1f09 : 06 b1 1b 91 19 88 10 f9  eb
1f11 : c8 68 91 1b c0 06 90 f8  46
1f19 : 84 02 4c f9 1e 20 73 00  a1
1f21 : 20 a0 aa 4c 74 a4 20 3f  31
1f29 : ab e4 d3 b0 f9 60 a9 ff  9a
1f31 : a0 01 91 2b 20 33 a5 a5  99
1f39 : 22 a6 23 38 69 01 90 01  61
1f41 : e8 85 2d 86 2e 4c 74 a4  68
1f49 : 08 a9 1f a2 d8 20 cd bd  83
1f51 : 20 3f ab a5 2c a6 2b 20  95
1f59 : cd bd 20 3f ab a5 2e a6  e3
1f61 : 2d 20 cd bd 20 3f ab a9  c7
1f69 : 00 38 e5 2d 85 63 a9 a0  00
1f71 : e5 2e 85 62 20 d1 bd a9  f6
1f79 : 66 a0 e4 20 1e ab 4c 74  c6
1f81 : a4 20 2a 14 a5 14 85 2b  aa
1f89 : a5 15 85 2c 20 44 a6 4c  f7
1f91 : 74 a4 20 9b b7 c9 2c d0  ef
1f99 : 16 e0 04 b0 12 86 21 20  51
1fa1 : 9b b7 a4 21 c0 02 f0 0a  59
1fa9 : 8a 99 f4 0e 4c 74 a4 4c  92
1fb1 : 48 b2 8a 8d f6 0e 4c 74  a1
1fb9 : a4 a9 00 85 21 20 73 00  c3
1fc1 : c9 52 f0 06 c9 48 d0 e7  a2
1fc9 : e6 21 20 9b b7 8a a4 21  60
1fd1 : 99 20 d0 4c 74 a4 00 ff  a4
```

Listing zum Programm »Hypra-Ass«. Bitte beachten Sie die Eingabehinweise auf Seite 110 (Schluß)

Reassembler zu Hypra-Ass

Passend zum Assembler Hypra-Ass stellen wir Ihnen einen professionellen Reassembler vor, der aus einem Maschinenprogramm Quelltext erzeugt.

Der Reassembler (Listing) erzeugt aus einem Maschinenprogramm Quelltext, der mit Hypra-Ass editiert, verändert und wieder assembliert werden kann. Der vollständig in Maschinensprache geschriebene Reassembler belegt den Speicherplatz von $C000 bis $CB00, kann aber mit dem SMON in jeden anderen Bereich verschoben werden. Neben dem eigentlichen Reassembler stehen noch einige Basic-Befehle zur Verfügung, mit denen zum Beispiel Einsprungpunkte im Quelltext durch ein Label markiert werden können. Es läßt sich auch vorherbestimmen, ob der Reassembler selbständig nach Tabellen suchen soll oder nicht. Weiterhin läßt sich der Aufbau des Quelltextes in einigen Punkten mitbestimmen. Alle dazu nötigen Informationen werden dem Reassembler in einem kleinen Basic-Informationsprogramm mitgeteilt. Es stehen dafür drei neue Basic-Befehle zu Verfügung:

←P adresse: Mit diesem Befehl lassen sich Einsprungpunkte im Quelltext durch ein Label markieren. So sind Adressen, die mit SYS angesprungen werden, im Quelltext leichter auffindbar.

←T adresse, adresse: Mit diesem Befehl teilen Sie dem Reassembler die Lage von Tabellen mit. Die erste Adresse zeigt auf das erste und die zweite Adresse auf das letzte Byte der Tabelle. Tabellen werden vom Reassembler nicht reassembliert, sondern erscheinen im Quelltext in Form eines Hex-Dumps (siehe Bild 1; Zeile 190 und Bild 2 Zeile 230 bis 310).

←E (byte): Der »E«-Befehl startet den Reassembler und steht am Ende des Informationsprogramms. Es wird nun aus einem Maschinenprogramm ein Quelltext erzeugt, der im Basic-Speicher abgelegt und anschließend wie ein normales Basic-Programm gespeichert oder editiert und mit Hypra-Ass assembliert werden kann.

Der Aufbau des Quelltextes läßt sich geringfügig beeinflussen, indem hinter den »E«-Befehl eine Zahl zwischen 0 und

255 eingegeben wird. Bei dieser Zahl handelt es sich um ein sogenanntes Informations-Byte. Die einzelnen Bits dieses Informations-Bytes haben folgende Bedeutung:

Informations-Byte:	o	o	o	o	o	o	o	o
Bit:	7	6	5	4	3	2	1	0
Wertigkeit:	128	064	032	016	008	004	002	001

Um zum Beispiel Bit 1 und Bit 6 auf 1 zu setzen, sind die Wertigkeiten der einzelnen Bits zu addieren. In diesem Fall 2 + 64 = 66

Bit 0 gesetzt: Alle Zeropage-Adressen (Adressen von $00 bis $FF) werden durch ein Label mit nur drei Buchstaben (normal fünf) markiert.

Bit 1 gesetzt: Nach den Befehlen RTS, RTI, BRK und JMP wird eine Kommentarzeile in den Quelltext eingefügt (Zeile 220 in Bild 2). Dadurch wird der Quelltext übersichtlicher.

Bit 2 gesetzt: Bei allen Befehlen mit unmittelbarer Adressierung (zum Beispiel LDA #$41) wird der Operand zusätzlich im ASCII-Format ausgegeben (LDY #$00 ;" . " — Zeile 140 und 160 in Bild 2), vorausgesetzt, er liegt zwischen 32 und 96 oder zwischen 160 und 224. Für den Fall, daß er außerhalb dieses Zahlenbereichs liegt, wird nur ein Punkt ausgegeben.

Bit 3 gesetzt: Zwischen je zwei Tabellenzeilen wird eine Kommentarzeile eingefügt (Zeile 240, 260, 280, 300 in Bild 2). Dieses erhöht die Übersichtlichkeit.

Bit 4 gesetzt: Der ASCII-Ausdruck wird bei Tabellen unterdrückt.

Bit 5 gesetzt: Ist dieses Bit gesetzt, werden externe Label und Tabellenlabel speziell gekennzeichnet (Zeile 100 und 230 in Bild 2). Tabellen wird ein »T« vorangestellt (zum Beispiel TLC000) und externen Label (Label die außerhalb des zu reassemblierenden Bereichs liegen) ein »E« (zum Beispiel ELC000).

Bit 6 gesetzt: Ist das Bit 6 gesetzt, sucht der Reassembler selbständig nach Tabellen. Es wird kein Quelltext, sondern ein Basic-Informationsprogramm generiert, das die Start- und Endadressen aller gefundenen Tabellen enthält. Dieses kann mit LIST oder — wenn Hypra-Ass geladen wurde — mit /E geLISTet und geändert werden.

Bit 7 gesetzt: Der Reassembler reassembliert die Speicherinhalte, die sich unter dem ROM im RAM befinden. Dadurch ist es möglich, Programme zu reassemblieren, die sich unter dem Basic-Interpreter oder Betriebssystem befinden.

Aus den drei neuen Basic-Befehlen setzt sich jedes Informationsprogramm zusammen. Es wird mit folgender Befehls-Sequenz im Direktmodus gestartet:
SYS 49152, anfadr, endadr+1:RUN
anfadr = Anfangsadresse des Maschinenprogramms, das reassembliert werden soll.
endadr = Endadresse des Maschinenprogramms, das reassembliert werden soll.

Um zum Beispiel den Reassembler durch sich selbst reassemblieren zu lassen, gehen Sie wie folgt vor:
1. Reassembler laden mit LOAD"REASS",8,1
2. NEW <RETURN> eingeben
3. Folgendes Basic-Informationsprogramm eingeben:
20←P $C000 ;kennzeichnet die Adresse $C000 durch ein Label
30←T $C813,$CAFF ;definiert eine Tabelle im Bereich von $C813 bis $CAFF
40←E 15 ;Startet den Reassembler und setzt die Bits 0 bis 3
4. SYS 49152,$C000,$CB00:RUN <RETURN>
im Direktmodus eingeben.

Die SYS-Zeile, mit der das Informationsprogramm gestartet wird, teilt dem Reassembler mit, daß das zu reassemblierende Maschinenprogramm im Bereich von $C000 bis $CAFF ($CB00 - 1) liegt. In Zeile 20 trifft der Reassembler auf den »P«-Befehl, der dazu auffordert, die Adresse $C000 durch ein Label zu markieren. Der »T«-Befehl in Zeile 30 definiert eine Tabelle im Bereich $C813 bis $CAFF und der »E«-Befehl in Zeile 40 startet schließlich den Reassembler.

In weniger als 8 Sekunden wird nun ein etwa 17 KByte langer Quelltext erzeugt, der mit LIST oder — wenn Hypra-Ass geladen und gestartet wurde — mit dem /E-Befehl geLISTet und mit RUN assembliert werden kann.

Wie Sie sicherlich schon bemerkt haben, verarbeitet der Reassembler nicht nur Dezimal-, sondern auch Hexadezimalzahlen. Eine Hexadezimalzahl beginnt mit einem Dollar-Zeichen ($), dem genau vier Hex-Ziffern folgen müssen. Beispiel:
$0073, $C000, $FFFF

Besonderheiten

1. Der Reassembler arbeitet ausgezeichnet mit Hypra-Ass zusammen. Dabei ist es jedoch übersichtlicher, das Informationsprogramm ohne Leerzeichen einzugeben, weil Hypra-Ass nach dem ersten Leerzeichen einen Tabulator einfügt. Das Aussehen des Quelltextes würde dadurch verunstaltet. Gestartet wird das Informationsprogramm wie gewohnt mit RUN. (Vergessen Sie nicht den Minusstrich vor jeder Zeile, wenn Hypra-Ass geladen ist.)

2. Aus programmtechnischen Gründen kann es vorkommen, daß der Reassembler im ersten Pass ein Maschinenprogramm anders reassembliert als im zweiten. Dadurch können in Pass 2 Sprungadressen im Maschinenprogramm auftauchen, die in Pass 1 nicht gefunden wurden und deshalb auch im Quelltext nicht durch ein Label markiert werden. Der Reassembler ersetzt in diesem Fall die Sprungadresse nicht durch ein Label, sondern stellt sie als Hex-Zahl im Quelltext dar. An die entsprechende Zeile werden 3 Fragezeichen angehängt.

3. 3-Byte-Befehle, die bei der Assemblierung als 2-Byte-Befehle interpretiert werden (BIT $A9 $00 = .BY $2C LDA #$00), werden nicht reassembliert. Statt dessen werden die 3 Byte mit vorangestelltem .BY-Pseudoopcode in den Quelltext eingefügt. Der reassemblierte Befehl wird aber als Kommentar an die entsprechende Zeile angefügt.

4. Es ist möglich, ein Programm so zu reassemblieren, als ob es in einem anderen Bereich läge. Dazu ist an den SYS-Befehl eine weitere Adresse anzuhängen:
SYS 49152, anfadr, endadr, get
anfadr und endadr geben die Anfangs- und Endadresse des Bereichs an, in dem das Maschinenprogramm liegen soll. get gibt die Anfangsadresse des Bereichs an, in dem das Programm tatsächlich liegt.

So kann man zum Beispiel die Kopie der CHRGET-Routine ab $E3A2 reassemblieren, als ob sie im Bereich von $0073 bis $008A liegen würde. Dazu ist im Direktmodus folgende Zeile einzugeben:
SYS 49152, $0073, $008A, $E3A2:←E
Da keine Tabellen in diesem Bereich liegen, kann auf ein Informationsprogramm verzichtet werden.

5. Es ist möglich, während der Reassemblierung den erzeugten Quelltext auf Diskette zu schreiben. Dadurch bleibt der Basic-Speicher für andere Programme frei. Dazu ist vor dem SYS-Befehl, mit dem die Start- und Endadresse übergeben wird, ein Programmfile zu öffnen. Mit dem Befehl CMD wird die Ausgabe auf das entsprechende Gerät umgeleitet. Das könnte wie folgt aussehen:
OPEN 1,8,1,"NAME,P,W":CMD 1:SYS 49152,$C000, $CB00:←E 64
Mit dem OPEN-Befehl wird ein Programmfile mit dem Namen »Name« zum Schreiben geöffnet. Der CMD-Befehl leitet die Ausgabe auf das Gerät mit der Gerätenummer 8 um (Disketten-Laufwerk). Der SYS-Befehl startet schließlich den Reassembler, dem durch den »E«-Befehl noch mitgeteilt wird, daß kein Quelltext, sondern ein Informationsprogramm erstellt werden soll. Das Informationsprogramm wird unter dem Namen »Name« auf Diskette gespeichert.

Vorsicht! Dieser Programmteil ist nicht gegen Fehlbedie-

nung abgesichert. So führt eine nicht eingelegte Diskette zum Absturz des Systems. In einem solchen Fall ist ein RESET auszulösen. Hypra-Ass kann anschließend mit SYS 2168 neu gestartet werden. Außerdem sollte nach jedem Speichern die RUN/STOP-RESTORE-Taste gedrückt werden.

Fehlermeldungen

SYNTAX ERROR: Ein Basic-Befehl wurde falsch eingegeben oder eine Hex-Zahl besteht aus weniger als 4 Hex-Ziffern.

OUT OF MEMORY: Es steht zu wenig Speicherplatz für den Quelltext zur Verfügung oder im Maschinenprogramm kommen mehr als 2700 verschiedene Label vor.

ILLEGAL QUANTITY: Vor einer Hex-Zahl fehlt das Dollar-Zeichen ($) oder das Tabellenende liegt vor dem Tabellenanfang oder die Tabellen überschneiden sich oder die angegebene Adresse liegt nicht im Maschinenprogramm.

TYPE MISMATCH: In einer Hex-Zahl stehen falsche Hex-Ziffern. Die Adresse, die schon als Einsprungpunkt markiert wurde, darf nicht als Tabellenanfang oder -ende angegeben werden. Im Informationsprogramm darf keine Adresse doppelt vorkommen.

Verschieben des Reassemblers

Der Reassembler benutzt in der Zeropage verschiedene Speicherzellen als Kurzzeitspeicher. Der Langzeitspeicher dagegen liegt unter dem Betriebssystem ($E000 bis $FFFF). Dort befindet sich auch ab Adresse $E028 die Label-Tabelle. In den Langzeitspeicher sollte nicht hineinge-POKEt werden.

Der Reassembler kann mit SMON ohne Schwierigkeiten im Speicher verschoben werden. Um den Reassembler nach $9000 zu verschieben, sind folgende SMON-Befehle einzugeben:

W C000 CB00 9000
V C000 CB00 9000 9000 9813

Beispiel zu den Basic-Erweiterungen

Laden Sie Hypra-Ass, starten Sie ihn und laden anschließend den Reassembler. Geben Sie NEW und danach im Direktmodus

SYS 49152,$1000,$1FD7:←E 64 <RETURN>

ein. Der Reassembler bekommt durch den SYS-Befehl die Start- und Endadresse des Maschinenprogramms mitgeteilt. Der »E«-Befehl setzt Bit 6 des Informations-Bytes und startet den Reassembler. Das gesetzte Bit 6 bewirkt, daß kein Quelltext, sondern ein Informationsprogramm erstellt wird. LISTen Sie das Programm mit /E. Sie sehen eine Reihe von »T«-Befehlen und zum Schluß einen »E«-Befehl. Schreiben Sie hinter diesen Befehl die Zahl 32, drücken die RETURN-Taste und geben folgende Zeile im Direktmodus ein:

OPEN 1,8,1,"REASS DEMO,P,W":CMD 1:SYS 49152,$1000,$1FD7:GOTO 100 <RETURN>

Mit dem OPEN-Befehl wird ein Programmfile mit dem Namen »REASS DEMO« zum Schreiben geöffnet. Der nachfolgende CMD-Befehl leitet die Ausgabe des Quelltextes auf dieses File um. Durch den SYS-Befehl wird dem Reassembler die Start- und Endadresse des Maschinenprogramms mitgeteilt. Der GOTO-Befehl startet schließlich das Informationsprogramm (der RUN-Befehl darf dazu nicht benutzt werden, da er das geöffnete File schließen würde). Die »T«-Befehle im Informationsprogramm werden ausgeführt, bis der »E«-Befehl Bit 5 setzt und den Reassembler startet. Das gesetzte fünfte Bit bewirkt, daß externe Label und Tabellenlabel speziell gekennzeichnet werden. Das so auf Diskette erzeugte Programmfile kann mit LOAD "REASS DEMO",8 geladen, geLISTet, editiert und assembliert werden.

(Martin Wehner/ah)

```
hypra-ass  assemblerlisting:

               10   -.li 1,4,7
;********************************
;*   Ausgabe eines Textes auf   *
;*        dem Bildschirm        *
;********************************
               60   -.ba $8000 ;Startadresse = $8000
;
               80   -.eq ausgabe = $ffd2 ;Dies ist ein externes Label
;
8000 a000    :100  -anfang    ldy #0          ;Schleifenzaehler auf 0 setzen
8002 b91080  :110  -loop      lda text,y      ;Zeichen holen
8005 c923    :120  -          cmp #"#"        ;mit Endekennzeichen vergleichen
8007 f006    :130  -          beq ende
8009 20d2ff  :140  -          jsr ausgabe     ;Zeichen ausgeben
800c c8      :150  -          iny             ;Schleifenzaehler + 1
800d d0f3    :160  -          bne loop
800f 60      :170  -ende      rts             ;Ende und zurueck ins Basic
;
              190  -text      .tx "Dies ist ein Beispieltext#"
```

Bild 1. Beispielprogramm zum »Reassembler« (Original Quelltext erstellt mit Hypra-Ass)

```
100  -            .eq elffd2=$ffd2
110  -;
120  -            .ba $8000
130  -;
140  -18000       ldy #$00         ; "."
150  -18002       lda t18010,y
160  -            cmp #$23         ; "#"
170  -            beq 1800f
180  -            jsr elffd2
190  -            iny
200  -            bne 18002
210  -1800f       rts
220  -;
230  -t18010      .by $c4,$49,$45,$53,$20,$49,$53,$54; "Dies ist"
240  -;
250  -            .by $20,$45,$49,$4e,$20,$c2,$45,$49; " ein Bei"
260  -;
270  -            .by $53,$50,$49,$45,$4c,$54,$45,$58; "spieltex"
280  -;
290  -            .by $54          ; "t"
300  -;
310  -t18029      .by $23          ; "#"
```

Bild 2. Reassembler-Quelltext zum Beispielprogramm aus Bild 1

```
programm : reass             c000 cb01
--------------------------------------

c000 : 20 4e c1 20 53 e4 ad 05  d9
c008 : c8 8d 02 03 ad 06 c8 8d  c1
c010 : 03 03 ad ff c7 8d 0a 03  17
c018 : ad 00 c8 8d 0b 03 20 53  99
c020 : c7 8d 09 e0 8c 08 e0 8d  b4
c028 : 01 e0 8c 00 e0 8d 03 e0  05
c030 : 8c 02 e0 48 98 48 20 53  f1
c038 : c7 8d 05 e0 8c 04 e0 20  d0
c040 : 79 00 f0 09 20 53 c7 8d  ed
c048 : 01 e0 8c 00 e0 68 85 14  6c
c050 : 68 85 15 a9 ff a2 02 9d  4e
c058 : 28 e0 ca 10 fa ad 02 c8  5c
c060 : 8d 08 03 ad 03 c8 8d 09  27
c068 : 03 20 d1 c0 20 d7 c0 4c  64
c070 : 1e c1 a9 00 85 0d 20 73  01
c078 : 00 b0 03 4c f3 bc 20 13  e6
c080 : b1 90 03 4c 28 af c9 24  33
c088 : f0 03 4c 9a ae 20 a0 c0  50
c090 : 85 62 20 a0 c0 85 63 a2  6d
c098 : 90 38 20 49 bc 4c 73 00  72
c0a0 : 20 bf c0 20 b5 c0 0a 0a  72
c0a8 : 0a 0a 85 14 20 bf c0 20  de
c0b0 : b5 c0 05 14 60 c9 3a 90  e8
c0b8 : 02 69 08 29 0f 38 60 20  0b
c0c0 : 73 00 90 12 c9 41 90 04  8b
c0c8 : c9 47 90 0a a2 16 6c 00  27
c0d0 : 03 78 a9 35 85 01 60 a0  43
c0d8 : 00 20 df c0 90 df c8 a5  6f
c0e0 : 14 d9 02 e0 c8 a5 15 f9  7f
c0e8 : 02 e0 60 20 73 00 20 56  db
c0f0 : c7 a2 80 20 36 c1 20 53  c5
c0f8 : c7 20 d1 c0 20 3c c5 b0  b8
c100 : 1d 20 09 c1 a2 00 4c 36  70
c108 : c1 20 57 c6 a0 00 b1 59  0b
c110 : f0 14 10 f5 c8 a5 14 d1  7b
c118 : 59 c8 a5 15 f1 59 b0 06  9a
c120 : a9 37 85 01 58 60 a2 0f  18
c128 : 20 20 c1 6c 00 03 20 73  d6
c130 : 00 20 56 c7 a2 02 20 69  5c
c138 : c0 20 d1 c0 20 27 c5 20  27
c140 : cd c5 d0 e4 a1 59 30 d8  18
c148 : 20 20 c1 4c ea a7 a0 0b  f7
c150 : 20 d1 c0 b9 00 03 48 b9  6d
c158 : 10 e0 99 00 03 68 99 10  39
c160 : e0 88 10 ef 4c 20 c1 20  93
c168 : 4e c1 6c 02 03 20 cd c1  de
c170 : 20 bc c1 b0 f8 a0 00 8c  22
c178 : af 02 2c a8 02 30 03 20  36
c180 : 20 c1 b1 5d aa 20 d1 c0  0d
c188 : e6 5d d0 02 e6 5e e6 5f  4d
c190 : d0 02 e6 60 a5 5f 85 14  bb
c198 : 29 3f d0 11 20 20 c1 a5  0c
c1a0 : 9a c9 08 b0 05 a9 c0 20  18
c1a8 : d2 ff 20 d1 c0 a5 60 85  82
c1b0 : 15 20 d7 c0 90 06 a9 80  c4
c1b8 : 8d af 02 60 a0 01 a5 5f  11
c1c0 : d1 fd c8 a5 60 f1 fd 90  25
c1c8 : 03 ee af 02 60 18 a5 fd  c8
c1d0 : 69 03 85 fd 90 02 e6 fe  8f
c1d8 : 60 20 70 c1 8e a9 02 bd  56
```

Listing zum »Reassembler«. Bitte beachten Sie die Eingabehinweise auf Seite 110.

```
c1e0 : ff c8 85 5c bd ff c9 8d   4e
c1e8 : b0 02 f0 17 b0 4f 30 4d   99
c1f0 : 29 0f 4a b0 1e 4a b0 0c   59
c1f8 : 20 70 c1 8e aa 02 a2 00   d8
c200 : 8e ab 02 60 20 70 c1 8e   9a
c208 : aa 02 b0 32 20 70 c1 8e   cf
c210 : ab 02 60 20 70 c1 8a 30   78
c218 : 0e 18 65 5f 8d aa 02 a5   f9
c220 : 60 69 00 8d ab 02 60 49   c5
c228 : ff 85 5b e6 5b 38 a5 5f   6a
c230 : e5 5b 8d aa 02 a5 60 e9   1e
c238 : 00 8d ab 02 60 18 a9 81   9a
c240 : 69 00 8d b0 02 a9 02 85   a3
c248 : 5c 60 20 73 00 f0 04 c9   76
c250 : 5f f0 03 4c e7 a7 20 73   95
c258 : 00 c9 50 d0 03 4c 2e c1   3a
c260 : c9 54 d0 03 4c eb c0 c9   a3
c268 : 45 f0 05 a2 0b 6c 00 03   d5
c270 : a2 00 8e ae 02 20 73 00   7b
c278 : f0 03 20 9e b7 8e a8 02   5c
c280 : 20 d1 c0 20 3e c7 20 27   ae
c288 : c5 20 bc c1 b0 1f 20 d9   fd
c290 : c1 ad b0 02 30 10 29 0f   db
c298 : f0 0c c9 08 f0 08 a2 01   de
c2a0 : 20 bf c5 20 ca c5 ad af   06
c2a8 : 02 f0 e3 30 17 a0 00 b1   fb
c2b0 : fd 10 db 20 70 c1 90 fb   ff
c2b8 : a0 00 b1 fd d0 f5 20 70   a2
c2c0 : c1 4c a6 c2 20 f5 c7 20   ba
c2c8 : 3e c7 20 d1 c7 a5 2b 85   8d
c2d0 : fb a5 2c 85 fc 20 75 c7   90
c2d8 : a0 00 b1 fd c9 ff f0 35   6f
c2e0 : 20 2f c7 20 d7 c0 90 27   a2
c2e8 : 20 1f c6 20 78 c7 a2 00   9e
c2f0 : 20 a4 c6 ad a8 02 29 20   49
c2f8 : f0 05 a9 45 20 34 c6 20   7d
c300 : b6 c6 a9 3d 20 34 c6 a9   3d
c308 : 24 20 b8 c6 20 70 c7 20   28
c310 : cd c1 4c d8 c2 20 3e c7   a2
c318 : 20 6d c7 20 1f c6 20 78   7e
c320 : c7 a2 01 20 a4 c6 a5 5f   53
c328 : 8d aa 02 a5 60 8d ab 02   65
c330 : a9 24 20 b8 c6 20 70 c7   c9
c338 : 20 6d c7 a0 00 b1 fd 30   fb
c340 : 10 20 2f c7 a0 00 20 df   6f
c348 : c0 b0 06 20 cd c1 4c 3b   78
c350 : c3 20 bc c1 90 03 4c 91   00
c358 : c4 20 1f c6 20 d9 c1 ad   00
c360 : ae 02 d0 3a ad b0 02 c9   87
c368 : 81 d0 33 ad af 02 d0 2e   7e
c370 : a8 a9 80 91 57 c8 8c ae   8a
c378 : 02 ad 08 e0 91 57 c8 ad   c1
c380 : 09 e0 91 57 20 6f c6 b1   45
c388 : fd 8d 09 e0 91 57 88 b1   04
c390 : fd 8d 08 e0 91 57 88 a9   bb
c398 : 00 91 57 20 62 c5 20 78   00
c3a0 : c7 ad b0 02 30 1c 29 0f   51
c3a8 : f0 69 c9 06 d0 12 ad ab   2c
c3b0 : 02 d0 0f a2 02 20 a4 c6   74
c3b8 : a9 83 4c 70 c4 4c 53 c4   c9
c3c0 : b0 fb 20 a2 c6 ad b0 02   6b
c3c8 : 30 f0 29 10 f0 05 a9 28   eb
c3d0 : 20 34 c6 20 bf c5 20 75   56
c3d8 : c5 20 b8 c6 ad b0 02 29   6f
c3e0 : 70 f0 33 c9 10 d0 08 a9   c9
c3e8 : 29 20 34 c6 4c 16 c4 c9   23
c3f0 : 40 90 11 f0 05 a9 29 20   5d
c3f8 : 34 c6 a9 2c a2 59 20 30   55
c400 : c6 4c 16 c4 a9 2c a2 58   42
c408 : 20 30 c6 ad b0 02 29 10   87
c410 : 4c e1 c3 20 a2 c6 a5 5c   f1
c418 : c9 37 d0 06 20 75 c7 20   7f
c420 : a2 c6 ad af 02 30 2a 20   11
c428 : 70 c7 a5 5c c9 38 90 19   44
c430 : ad ae 02 d0 0a a5 5f 8d   35
c438 : 08 e0 a5 60 8d 09 e0 ad   26
c440 : a8 02 29 02 f0 03 20 6d   f6
c448 : c7 ad af 02 d0 43 4c 59   1d
c450 : c3 30 58 20 a2 c6 a9 23   93
c458 : 20 34 c6 ad aa 02 8d a9   3e
c460 : 02 a2 71 ad a8 02 29 04   0d
c468 : f0 02 a2 81 8a 8d b0 02   0e
c470 : 29 03 85 5b aa bd a8 02   27
c478 : 9d 1f e0 ca d0 f7 20 ec   5d
c480 : c6 ad b0 02 c9 70 b0 8e   89
c488 : 20 75 c7 20 a2 c6 4c d3   92
c490 : c3 20 2f c7 20 1f c6 20   7e
c498 : 63 c6 a0 00 8c ae 02 b1   30
c4a0 : fd f0 70 30 2e 20 b6 c6   84
c4a8 : 4c 5c c3 a2 00 a9 ff 81   b8
c4b0 : 57 ad ae 02 f0 03 20 4a   06
c4b8 : c5 20 7d c7 20 33 c6 20   dd
c4c0 : 30 c6 20 20 c1 a5 b8 20   cc
c4c8 : c3 ff 20 b5 ab 6a 85 9d   a9
c4d0 : 4c ab e1 20 e6 c7 20 3d   16
c4d8 : c2 ad a8 02 29 10 f0 05   bc
c4e0 : a9 71 8d b0 02 a0 00 84   e9
c4e8 : 5b 20 70 c1 a4 5b 8a 99   2a
c4f0 : 20 e0 e6 5b b0 0a c0 07   12
c4f8 : d0 ef 20 16 c6 4c d6 c4   3e
c500 : a0 00 b1 fd f0 04 09 e0   e1
c508 : 91 fd 20 e6 c6 20 5c c7   eb
c510 : 4c 16 c4 20 e6 c7 20 70   aa
c518 : c1 8e a9 02 20 3d c2 4c   5b
c520 : 9e c3 20 3c c5 90 10 ad   ac
c528 : 08 c8 85 59 ad 09 c8 85   72
c530 : 5a 4c 37 c5 20 57 c6 20   4f
c538 : 3c c5 90 f8 a0 02 b1 59   2e
c540 : c5 15 d0 05 88 b1 59 c5   6c
c548 : 14 60 38 a5 57 e9 03 85   2b
c550 : 57 b0 02 c6 58 a0 01 ad   43
c558 : 04 e0 91 57 c8 ad 05 e0   eb
c560 : 91 57 38 a0 01 b1 57 e9   8e
c568 : 01 91 57 c8 b1 57 e9 00   9e
c570 : 91 57 4c 6f c6 20 22 c5   2f
c578 : d0 08 a0 00 b1 59 c9 ff   81
c580 : 90 07 a9 37 85 5c a9 24   0f
c588 : 60 ad a8 02 29 20 f0 1c   b9
c590 : 20 d7 c0 90 1a a9 45 20   22
c598 : 34 c6 4c ac c5 20 57 c6   20
c5a0 : b1 59 f0 13 c9 ff f0 04   05
c5a8 : c9 80 d0 f1 a9 4c 60 a0   e3
c5b0 : 00 b1 59 c9 80 d0 e9 a9   a2
c5b8 : 54 20 34 c6 4c ac c5 ad   9f
c5c0 : aa 02 85 14 ad ab 02 85   9a
c5c8 : 15 60 20 22 c5 f0 44 86   5c
c5d0 : 61 a5 59 85 5b a5 5a 85   62
c5d8 : 5c a0 02 b1 5b b6 61 99   5f
c5e0 : 61 00 8a 91 5b 88 10 f3   38
c5e8 : 18 a5 5b 69 03 85 5b 90   c2
c5f0 : 02 e6 5c e0 ff d0 e2 a0   ec
c5f8 : 01 a5 14 91 59 c8 a5 15   a0
c600 : 91 59 a5 5c cd 0c c8 90   b4
c608 : 07 a5 5b cd 0b c8 b0 7d   27
c610 : a2 00 60 a2 0e 60 20 e6   51
c618 : c6 20 5c c7 20 70 c7 18   d3
c620 : ad ac 02 69 0a 8d ac 02   95
c628 : 90 03 ee ad 02 ae ad 02   fb
c630 : 20 34 c6 8a 2c a8 02 70   5e
c638 : 66 a4 9a c0 08 b0 53 a0   44
c640 : 00 91 fb e6 fb a4 fb 84   c2
c648 : 2d d0 02 e6 fc a4 fc 84   2d
c650 : 2e c4 38 b0 38 90 48 18   5e
c658 : a5 59 69 03 85 59 90 02   ce
c660 : e6 5a 60 a0 02 b1 fd 91   68
c668 : 57 88 10 f9 4a b0 f3 2c   99
c670 : a8 02 50 0b 18 a5 57 69   6e
c678 : 03 85 57 90 02 e6 [illegible] [illegible]   2a
c680 : 58 cd 12 c8 90 dc a5 57   92
c688 : cd 11 c8 90 d5 a2 10 4c   6d
c690 : 28 c1 48 a8 20 20 c1 98   fb
c698 : 20 d2 ff 20 d1 c0 68 a0   2b
c6a0 : 00 60 a6 5c bd 4b c8 20   9f
c6a8 : 34 c6 bd 87 c8 20 34 c6   8b
c6b0 : bd c3 c8 4c 34 c6 a9 4c   c3
c6b8 : 20 34 c6 ad a8 02 29 01   9b
c6c0 : f0 05 ad ab 02 f0 06 ad   2f
c6c8 : ab 02 20 d0 c6 ad aa 02   1f
c6d0 : 48 4a 4a 4a 4a 20 db c6   bc
c6d8 : 68 29 0f c9 0a 90 02 69   d2
c6e0 : 06 69 30 4c 34 c6 20 78   1b
c6e8 : c7 20 a2 c6 a2 00 f0 05   39
c6f0 : a9 2c 20 34 c6 a9 24 20   c8
c6f8 : 34 c6 bd 20 e0 20 d0 c6   e3
c700 : e8 e4 5b 90 eb ad b0 02   36
c708 : 10 33 20 75 c7 20 2b c7   22
c710 : a2 00 bd 20 e0 a8 29 7f   1d
c718 : c9 20 90 04 c9 60 90 02   7c
c720 : a0 2e 98 20 34 c6 e8 e4   e8
c728 : 5b 90 e7 a9 22 d0 4b a0   11
c730 : 00 20 34 c7 c8 b1 fd 99   8c
c738 : 13 00 99 a9 02 60 ad 08   d1
c740 : c8 85 fd ad 09 c8 85 fe   eb
c748 : a2 04 bd ff df 95 5c ca   0d
c750 : d0 f8 60 20 fd ae 20 8a   a3
c758 : ad 4c f7 b7 ad a8 02 29   9b
c760 : 08 f0 da 20 70 c7 20 1f   9f
c768 : c6 a9 3b d0 0d 20 66 c7   e7
c770 : a9 20 20 34 c6 20 69 c7   4b
c778 : a9 20 4c 34 c6 2c a8 02   3f
c780 : 8e a8 02 50 60 20 d1 c7   cb
c788 : 20 f5 c7 20 75 c7 20 1f   ed
c790 : c6 a9 5f 20 34 c6 b1 57   f6
c798 : c9 ff f0 30 c9 80 f0 0c   1f
c7a0 : a9 50 20 b5 c7 98 20 34   5a
c7a8 : c6 4c 8b c7 a9 54 20 b5   99
c7b0 : c7 a9 2c d0 ed 20 34 c6   af
c7b8 : c8 b1 57 8d aa 02 c8 b1   22
c7c0 : 57 8d ab 02 a9 24 20 b8   b7
c7c8 : c6 4c 74 c6 a9 45 20 34   58
c7d0 : c6 a9 5a 8d ac 02 a2 00   18
c7d8 : 8e ad 02 ad 0e c8 85 57   5f
c7e0 : ad 0f c8 85 58 60 ad a8   88
c7e8 : 02 29 20 f0 05 a9 54 20   d4
c7f0 : 34 c6 4c b6 c6 a5 9a c9   09
c7f8 : 08 90 ea 4c 34 c6 2c 72   9c
c800 : c0 2c 4a c2 2c 67 c1 2c   1f
c808 : 28 e0 2c f0 f7 2c 00 f8   9c
c810 : 2c f0 ff 2e 2e 2e 20 48   df
c818 : 59 50 52 41 2d 52 45 2c   29
c820 : 20 50 55 42 4c 49 53 48   f3
c828 : 45 44 20 42 59 20 36 34   b8
c830 : 27 45 52 2c 20 28 43 29   b7
c838 : 20 31 39 38 35 20 42 59   56
c840 : 20 4d 2e 20 57 45 48 4e   f4
c848 : 45 52 20 2e 2e 2e 41 41   60
c850 : 41 42 42 42 52 42 42 43   52
c858 : 43 53 43 43 45 49 49 50   e2
c860 : 4c 44 42 44 43 4e 43 44   24
c868 : 4c 53 54 42 54 4f 53 42   4d
c870 : 49 50 4a 4c 4c 50 50 53   2d
c878 : 42 43 53 52 53 53 54 54   45
c880 : 54 54 3f 42 4a 52 52 45   22
c888 : 42 42 44 4e 53 43 43 45   ad
c890 : 4f 50 56 4c 4c 45 4d 50   eb
c898 : 4f 4e 4e 4c 44 45 56 45   7e
c8a0 : 4c 4f 50 45 44 42 59 4d   a7
c8a8 : 41 52 54 49 4e 4c 53 44   6e
c8b0 : 53 48 48 45 4e 4c 45 4f   dd
c8b8 : 54 54 41 53 58 58 3f 52   db
c8c0 : 4d 54 54 51 41 59 43 44   eb
c8c8 : 4c 43 53 51 52 4c 53 43   10
c8d0 : 49 43 50 58 52 43 59 50   1f
c8d8 : 41 43 43 58 44 50 59 59   76
c8e0 : 58 43 41 49 58 41 59 54   f1
c8e8 : 58 41 52 59 52 41 50 44   9a
c8f0 : 45 56 49 4c 41 58 59 58   29
c8f8 : 41 53 3f 4b 50 49 53 38   29
c900 : 22 02 02 02 22 05 02 2b   8d
c908 : 22 05 02 02 22 05 02 0a   d4
c910 : 22 02 02 02 22 05 02 0c   5f
c918 : 22 02 02 02 22 05 02 27   9d
c920 : 04 02 02 24 04 30 02 14   1c
c928 : 04 30 02 24 04 30 02 20   54
c930 : 04 02 02 02 04 30 02 0e   dc
c938 : 04 02 02 02 04 30 02 3a   3c
c940 : 11 02 02 02 11 29 02 2a   ca
c948 : 11 29 02 39 11 29 02 17   26
c950 : 11 02 02 02 11 29 02 0d   a0
c958 : 11 02 02 02 11 29 02 3b   04
c960 : 03 02 02 02 03 09 02 26   f2
c968 : 03 09 02 39 03 09 02 0b   2e
c970 : 03 02 02 02 03 09 02 2f   14
c978 : 03 02 02 02 03 09 02 02   c2
c980 : 31 02 02 23 31 32 02 1c   7c
c988 : 02 35 02 23 31 32 02 06   c3
c990 : 31 02 02 23 31 32 02 1f   92
c998 : 31 36 02 02 31 02 02 28   21
c9a0 : 15 1d 02 28 15 1d 02 33   72
c9a8 : 15 21 02 28 15 1d 02 07   24
c9b0 : 15 02 02 28 15 1d 02 2e   eb
c9b8 : 15 34 02 28 15 1d 02 1b   e5
c9c0 : 0f 02 02 1b 0f 16 02 13   84
c9c8 : 0f 18 02 1b 0f 16 02 2d   cb
c9d0 : 0f 02 02 02 0f 16 02 19   7d
c9d8 : 0f 02 02 02 0f 16 02 10   73
c9e0 : 1e 02 02 10 1e 12 02 25   47
c9e8 : 1e 1a 02 10 1e 12 02 08   21
c9f0 : 1e 02 02 02 1e 12 02 2c   a3
c9f8 : 1e 02 02 02 1e 12 02 00   53
ca00 : 34 80 80 80 04 04 80 00   07
ca08 : 08 00 80 80 06 06 80 01   d5
ca10 : 54 80 80 80 24 24 80 00   3a
ca18 : 42 80 80 80 26 26 80 02   64
ca20 : 34 80 80 04 04 04 80 00   97
ca28 : 08 00 80 00 06 06 80 01   a6
ca30 : 54 80 80 80 24 24 80 00   5a
ca38 : 42 80 80 80 26 26 80 00   80
ca40 : 34 80 80 80 04 04 80 00   47
ca48 : 08 00 80 02 06 06 80 01   45
ca50 : 54 80 80 80 24 24 80 00   7a
ca58 : 42 80 80 80 26 26 80 00   a0
ca60 : 34 80 80 80 04 04 80 00   67
ca68 : 08 00 80 12 06 06 80 01   67
ca70 : 54 80 80 80 24 24 80 00   9a
ca78 : 42 80 80 80 26 26 80 80   c1
ca80 : 34 80 80 04 04 04 80 00   f7
ca88 : 80 00 80 06 06 06 80 01   7e
ca90 : 54 80 80 24 24 44 80 00   2f
ca98 : 42 00 80 80 26 80 80 08   83
caa0 : 34 08 80 04 04 04 80 00   db
caa8 : 08 00 80 06 06 06 80 01   26
cab0 : 54 80 80 24 24 44 80 00   4f
cab8 : 42 00 80 26 26 46 80 08   86
cac0 : 34 80 80 04 04 04 80 00   37
cac8 : 08 00 80 06 06 06 80 01   46
cad0 : 54 80 80 80 24 24 80 00   fa
cad8 : 42 80 80 80 26 26 80 08   30
cae0 : 34 80 80 04 04 04 80 00   57
cae8 : 08 00 80 06 06 06 80 01   66
caf0 : 54 80 80 80 24 24 80 00   1a
caf8 : 42 80 80 80 26 26 80 00   40
cb00 : ac ff 00 ff 00 ff 00 ff   ac
```

Listing zum »Reassembler« (Schluß)

SMON – komplett

Die Stärken dieses Super-Maschinensprache-Monitors sind hauptsächlich die mächtigen Such- und Trace-Befehle zum Austesten von Programmen in Maschinensprache. Neben dem kompletten Listing und der kompletten Anleitung mit einer Tabelle sämtlicher Funktionen und einer Tabelle wichtiger Einsprungadressen finden Sie zwei Erweiterungen, einen vollständigen Diskmonitor und einen Disassembler, der auch illegale Opcodes disassembliert. Ein Programm, mit dem auch Profis gerne arbeiten.

Ich kann mich noch gut an unsere ersten Schritte in Maschinensprache erinnern. Ausgerüstet mit einer Befehlsliste für den 6502 und einem in Basic geschriebenen »Mini-Monitor« entstanden Programme, die 3 und 5 addieren und das Ergebnis im Speicher ablegen konnten. Dazu mußten wir die Befehlcodes aus der Liste heraussuchen und dann in den Speicher »POKEn«. Jeder Sprung mußte von Hand ausgerechnet werden, jeder falsch herausgesuchte Befehl führte zum Programmabsturz. Der erste Disassembler – ein Programm zur Anzeige der Maschinenbefehle in Assemblersprache – war für uns die Offenbarung. Von nun an konnten wir Maschinenprogramme analysieren und daraus lernen. Zum Verständnis der Maschinensprache ist es nämlich noch weit mehr als bei anderen Sprachen wichtig, vorhandene Programme zu verstehen und sich dabei die wichtigsten Techniken anzueignen.

Mit der Zeit wuchsen unsere Ansprüche, ein Assembler mußte her, um die neugewonnenen Erkenntnisse auch auszuprobieren. Das war zuerst wieder ein Basic-Programm, langsam und wenig komfortabel, aber immerhin. Wir schrieben unsere ersten kleinen Routinen, vor allem, um vorhandene Maschinenprogramme unseren eigenen Wünschen anzupassen. Mit dem AMON für den VC 20 bekamen wir dann einen Monitor, der (fast) alle unsere Wünsche erfüllte. Als wir jedoch auf den C 64 umstiegen, mußten wir feststellen, daß es für diesen Computer nichts gab, das uns zufriedenstellen konnte. Der einzige Ausweg: selbst programmieren. So entstand im Laufe eines Jahres SMON. Ursprünglich hatten wir nur vor, die Funktionen von AMON für den C64 zu programmieren, aber dabei blieb es nicht. Immer neue Befehle und Routinen kamen hinzu, bis wir endlich zufrieden waren.

Was bietet SMON?

Zunächst ist alles enthalten, was zum »Standard« gehört: Memory-Dump, also die Anzeige des Speicherinhalts in Hex-Bytes, mit Änderungsmöglichkeiten, ein Disassembler mit Änderungsmöglichkeit sowie Routinen zum Laden, Abspeichern und Starten von Maschinenprogrammen. Darüber hinaus gibt es einen kleinen Direktassembler, der sogar Labels verarbeitet, Befehle zum Verschieben im Speicher mit und ohne Umrechnen der Adressen und Routinen zum Umrechnen von Hex-, Dezimal- und Binärzahlen. Der besondere Clou von SMON liegt aber zweifellos in seinen leistungsfähigen Suchroutinen und vor allem im Trace-Modus. Damit lassen sich Maschinenprogramme Schritt für Schritt abarbeiten und kontrollieren.

Der Monitor benötigt für alle Eingaben die hexadezimale Schreibweise, das heißt zu den Zahlen 1 bis 9 kommen noch die Buchstaben A (für dez. 10) bis F (für dez. 15) hinzu.

Bei der Eingabe von Adressen ist folgendes zu beachten: [ANFADR] bedeutet exakt die Startadresse, [ENDADR] bedeutet hierbei die erste Adresse hinter dem gewählten Bereich. Im Normalfall ist die Eingabe mit und ohne Leerzeichen zulässig. Beim Abweichen von dieser Regel wird darauf besonders verwiesen. Tippen Sie zuerst das Hauptprogramm (Listing 1) mit dem MSE ab. Befindet sich SMON auf Ihrer Diskette, kann er mit LOAD "SMON $C000",8,1 geladen und mit dem Befehl SYS 49152 gestartet werden. Geben Sie vor dem SYS-Befehl aber NEW ein, um einen späteren »OUT OF MEMORY« zu verhindern.

Assemblieren
A [ANFADR]
Assemblierung beginnt bei der angegebenen Adresse
Beispiel:
A 4000 Beginn bei Startadresse $4000

Nach Eingabe von »RETURN« erscheint auf dem Bildschirm die gewählte Adresse mit einem blinkenden Cursor. Die Befehle werden so eingegeben, wie sie der Disassembler zeigt: LDY #00 oder LDA 400E,Y und so weiter. »RETURN« schließt die Eingabe der Zeile ab. Bei fehlerhafter Eingabe springt der Cursor wieder in die Anfangsposition zurück. Ansonsten wird der Befehl disassembliert und nach Ausgabe der Hex-Bytes gelistet. Zur Korrektur vorhergehender Zeilen gehen Sie mit dem Cursor zur Anfangsposition (hinter die Adresse) zurück, schreiben den Befehl neu und gehen nach »RETURN« mit dem Cursor wieder in die letzte Zeile. Falls Ihnen bei Sprüngen (Branch-Befehl, JSR und JMP) die Zieladressen noch nicht bekannt sind, geben Sie einfach sogenannte »Label« ein.

Ein Label besteht aus dem Buchstaben »M« (für Marke) und einer zweistelligen Hex-Zahl von 01 bis 30.
Beispiel: BCC M01

Wenn Sie die Zieladresse für diesen Sprung erreicht haben, dann kennzeichnen Sie diese mit eben dieser »Marke«.
Beispiel: M01 LDY #00

Einzelne Bytes nimmt der Assembler an, indem Sie diese mit einem Punkt kennzeichnen: .00 oder .AB. In diesem Modus werden die Eingaben natürlich nicht disassembliert.

Nach Beendigung des Assemblierens geben Sie »F« ein. Danach sehen Sie alle Ihre Eingaben noch einmal aufgelistet und korrigieren dann bei Bedarf wie beim Disassembler (!) angegeben.

Probieren Sie einmal das folgende Beispiel:
A 4000
Der Assembler meldet sich mit: »4000« und einem blinkenden Cursor. Geben Sie nun ein (die Adressen erscheinen automatisch):

```
4000 LDY #00        4009 CPY #12
4002 LDA 400E,Y     400B BCC 4002
4005 JSR FFD2       400D BRK
4008 INY
```

Die folgenden Bytes werden wie beschrieben mit einem Punkt eingegeben. Sie werden nicht disassembliert.

400E .0D	4017 .54
400F .0D	4018 .20
4010 .53	4019 .53
4011 .4D	401A .55
4012 .4F	401B .50
4013 .4E	401C .45
4014 .20	401D .52
4015 .49	401E .0D
4016 .53	401F .0D

Drücken Sie anschließend »F«. Ihr Programm wird nochmal aufgelistet. Starten Sie es nun mit »G 4000«. Es erscheint ein Text auf dem Bildschirm - lassen Sie sich überraschen.

Disassemblieren

D [ANFADR,ENDADR]
disassembliert den Bereich von ANFADR bis ENDADR, wobei ENDADR nicht eingegeben werden muß. Wird keine Endadresse eingegeben, erscheint zunächst nur eine Zeile:

```
ADR      HEXBYTES   BEFEHL
4000     A0 00      LDY #00
```

Mit der SPACE-Taste wird der jeweils nächste Befehl in der gleichen Art und Weise gezeigt. Wünschen Sie eine fortlaufende Ausgabe, drücken Sie »RETURN«. Die Ausgabe wird dann so lange fortgesetzt, bis eine weitere Taste gedrückt wird oder bis ENDADR erreicht ist. Mit »RUN/STOP« springen Sie jederzeit in den Eingabemodus zurück.

Das Komma, das vor der Adresse auf dem Bildschirm erscheint, ist ein »hidden command« (verstecktes Kommando). Es braucht nicht eingegeben zu werden, da es automatisch beim Disassemblieren angezeigt wird. So ermöglicht es ein einfaches Ändern des Programms. Fahren Sie mit dem Cursor auf den zu ändernden Befehl und überschreiben Sie ihn mit dem neuen. Wenn Sie jetzt »RETURN« drücken, erkennt SMON das Komma als Befehl und führt ihn im Speicher aus. Achten Sie aber darauf, daß der neue Befehl die gleiche Länge (in Byte) hat und füllen Sie gegebenenfalls mit »NOPs« auf. Zur Kontrolle können Sie den geänderten Bereich noch einmal disassemblieren.

Lassen Sie als Beispiel einmal das Programm (siehe Befehl »A«) ab 4000 disassemblieren (»D 4000 4011«). Ändern Sie nun den ersten Befehl auf LDY #01. Die Änderung zeigt sich daran, daß die HEX-Bytes automatisch den neuen Wert annehmen. Starten Sie nun das Programm nochmals mit »G 4000«. Jetzt erscheint der Text mit nur einer Zeile Abstand auf dem Bildschirm.

Starten eines Maschinenprogramms (Go)

G [ADRESSE]
startet ein Maschinenprogramm, das bei ADRESSE beginnt. Das Programm muß mit einem BRK-Befehl abgeschlossen werden, damit ein Rücksprung in SMON erfolgen kann. Wird nach »G« keine Adresse eingegeben, benutzt SMON die, die mit dem letzten BRK erreicht worden ist und bei der Register-Ausgabe als PC auftaucht. Mit dem »R«-Befehl (siehe unten) werden die Register vorher auf gewünschte Werte gesetzt.

Memory-Dump

M [ANFADR ENDADR]
gibt die HEX-Werte des Speichers sowie die zugehörigen ASCII-Zeichen aus. Auch hier kann auf die Eingabe einer Endadresse verzichtet werden. Die Steuerung der Ausgabe entspricht der beim Disassemblieren.
Beispiel:

M 4000 gibt die Inhalte der Speicherstellen $4000 bis $4007 aus. Weiter geht es wie beim Disassemblieren mit SPACE oder RETURN. Die Bytes können ebenfalls durch Überschreiben geändert werden, allerdings nicht die ASCII-Zeichen. Verantwortlich dafür ist der Doppelpunkt, der am Anfang jeder Zeile ausgegeben wird, ein weiterer »hidden command«. Wenn Ihre Änderung nicht durchgeführt werden kann, weil Sie zum Beispiel versuchen, ins ROM zu schreiben, wird ein »?« als Fehlermeldung ausgegeben.

Registeranzeige

R zeigt den gegenwärtigen Stand der wichtigsten 6510-Register an: Programmzähler (PC), Status-Register (SR), Akkumulator (AC), X-Register (XR), Y-Register (YR), Stackpointer (SP). Außerdem werden die einzelnen Flags des Status-Registers mit 1 für »gesetzt« und 0 für »nicht gesetzt« angezeigt. Durch Überschreiben werden die Inhalte auf einen gewünschten Wert gesetzt. Die Flags können allerdings nicht einzeln verändert werden, sondern nur durch Überschreiben des Wertes von SR.

Exit

X springt ins Basic zurück. Alle Basic-Pointer bleiben erhalten. Sie können also zum Beispiel direkt im Programm fortfahren, wenn Sie zwischendurch mit SMON einige Speicherstellen kontrolliert haben.

Probieren Sie alle bisher beschriebenen Befehle in Ruhe aus und machen Sie sich mit SMON vertraut. Arbeiten Sie auch parallel den Kurs über Assemblerprogrammierung in dieser Ausgabe durch. Alle Beispiele dort sind auf SMON abgestimmt.

I/O-SET

IO 1 legt die Device-Nummer für LOAD und SAVE auf 1 (Kassette). Jedes Laden und Abspeichern erfolgt jetzt auf das angegebene Gerät. Die voreingestellte Device-Nummer ist 8 (für die Floppy also: IO 8). Wenn Sie nur mit der Floppy arbeiten, brauchen Sie diesen Befehl also nicht.

LOAD

L"name" lädt ein Programm vom angegebenen Gerät (wie oben beschrieben) an die Originaladresse in den Speicher. Die Basic-Zeiger bleiben bei diesem Ladevorgang unbeeinflußt, das heißt, sie werden nicht verändert.

Beispiel: Unser Monitor soll an seiner Originaladresse ($C000) im Speicher stehen. Also brauchen Sie ihn nur mit »L"SMON"« zu laden, damit er dort erscheint. Wenn Sie einmal ein Programm an eine andere als die Originaladresse laden wollen, dann bietet Ihnen SMON dazu folgende Möglichkeit: »L"name" ADRESSE« lädt ein Programm an die angegebene Adresse. Nehmen Sie doch bitte noch einmal unser letztes Test-Programm und geben es mit dem Assembler ab Adresse $4000 ein. Speichern Sie es mit »S"SUPERTEST" 4000 4023« ab und laden es dann
1. an die Originaladresse (L"SUPERTEST") und
2. an eine andere Adresse (mit L"SUPERTEST"5000 zum Beispiel nach $5000).

Schauen Sie sich danach mit dem Disassembler-Befehl beide Routinen einmal an. Sie werden feststellen, daß beide Programme zwar bis auf die BRANCH-Befehle gleich aussehen, daß das Programm in $5000 aber nicht funktionieren kann, da es eine falsche Adresse verwendet (5002 LDA 400E,Y). Ein anderes Beispiel dazu: Ein Autostart-Programm beginnt bei $0120, läßt sich aber in diesem Bereich nicht untersuchen, da dort der Prozessor-STACK (im Bereich von $0100 bis $01FF) liegt, der vom Prozessor selbständig verändert wird. Wenn Sie nun L"name" 4120 eingeben, befindet sich das Programm anschließend bei $4120 (nicht an der Originaladresse $0120) und Sie können es ohne Einschränkungen - von den falschen Absolut-Adressen abgesehen - disassemblieren.

SAVE

S"name", ANFADR ENDADR speichert ein Programm von ANFADR bis ENDADR-1 unter »name« auf die Floppy ab, da diese - wie wir ja inzwischen wissen - das voreingestellte Gerät ist. Wenn Sie auf Kassette abspeichern wollen, setzen Sie vorher mit »IO 1« die Device-Nummer auf 1.

Beispiel: S"SUPERTEST"4000 4020 speichert das Programm mit dem Namen »SUPERTEST« (es steht im Speicher

von $4000 bis $401F) auf Diskette ab. Bitte beachten Sie auch bei diesem Befehl, daß die Endadresse auf das nächste Byte hinter dem Programm gesetzt wird.

Printer-Set

P0 2 setzt die Primäradresse für den Drucker auf 2. Voreingestellt ist hier die 4 als Gerätenummer (zum Beispiel für Commodore-Drucker). Vielleicht haben Sie es ja schon bemerkt: Bei allen Ausgabe-Befehlen (wie D, M etc.) können Sie auch den Drucker ansprechen, wenn Sie das Kommando geshiftet eingeben. Die Ausgabe erfolgt dann gleichzeitig auf Bildschirm und Drucker. (Beachten Sie bitte die Änderung für die Druckerausgabe am Schluß des Artikels.)

Ein bißchen Rechnerei

Die folgende Befehlsgruppe enthält Befehle zur Zahlenumrechnung. Sie wissen ja: Der Mensch mit seinen zehn Fingern neigt eher zur dezimalen Rechenweise, aber der Computer bevorzugt das Binärsystem, weil er nur zwei Finger hat (siehe Netzstecker). Ein Kompromiß ist das Hexadezimalsystem, denn das versteht keiner von beiden. Um Verständnisschwierigkeiten mit Ihrem Liebling aus dem Weg zu gehen, haben Sie aber SMON.

Umrechnung Dez→Hex

(Dezimalzahl) rechnet die Dezimalzahl in die entsprechende Hexadezimalzahl um. Hierbei können Sie die Eingabe in beliebiger Weise vornehmen, da SMON Zahlen bis 65 535 umrechnet. Beispiel: #12, #144, #3456, #65533 und so weiter.

Umrechnung Hex→Dez

$ (Hexadezimalzahl) rechnet die Hexadezimalzahl in die entsprechende Dezimalzahl um. Die Eingabe muß hierbei zweistellig beziehungsweise vierstellig erfolgen. Ist diese Zahl kleiner als $100 (=255), wird zusätzlich auch der Binärwert ausgegeben.

Beispiel: $12, $0012, $0D, $FFD2 etc. In den ersten drei Beispielen erfolgt die Anzeige auch in binärer Form.

Umrechnung Binär→Hex,Dez

% (Binärzahl (achtstellig)) rechnet die Binärzahl in die entsprechenden Hexa- und Dezimalzahlen um. Bei diesem Befehl müssen Sie genau acht Binärzahlen eingeben. Falls Sie einmal versehentlich mehr eingeben sollten, werden nur die ersten acht zur Umrechnung herangezogen. Beispiel: %00011111, %10101011

Add-Sub

? 2340+156D berechnet die Summe der beiden vier (!)-stelligen Hex-Zahlen. Neben der Addition ist auch Subtraktion möglich.

Programme auf dem Rangierbahnhof

Occupy (Besetzen)

O (ANFADR ENDADR HEX-Wert) belegt den angegebenen Bereich mit dem vorgegebenen HEX-Wert. Beispiel: O 5000 8000 00 füllt den Bereich von $5000 bis $7FFF mit Nullen.

Man kann mit »OCCUPY« aber nicht nur Speicherbereiche löschen, sondern auch mit beliebigen Werten belegen. Häufig hat man das Problem, festzustellen, welcher Speicherplatz von einem Programm wirklich benutzt wird. Wir füllen den in Frage kommenden Bereich dann zuerst zum Beispiel mit »AA« und laden dann unser Programm. Probieren Sie bitte das folgende Beispiel: Füllen Sie den Speicherbereich von $3000 bis $6000 mit $AA und laden Sie dann unser SUPERTEST-Programm. Beim Disassemblieren können Sie erkennen, daß unser kleines Programm exakt zwischen vielen »AA« eingebettet ist.

Write

W (ANFADRalt ENDADRalt ANFADRneu) verschiebt den Speicherbereich von ANFADRalt bis ENDADRalt nach ANFADRneu ohne Umrechnung der Adressen! Unser kleines Testprogramm möge noch einmal als Beispiel dienen: W 4000 4020 6000 verschiebt das oben angesprochene Programm von $4000 nach $6000.

Hierbei werden weder die absoluten Adressen umgerechnet noch die Tabellen geändert. Letzteres ist sicherlich erwünscht, aber denken Sie daran, daß das verschobene Programm nun nicht mehr lauffähig ist, da die absoluten Adressen nicht mehr stimmen (zum Beispiel bei dem Befehl LDA 400E,Y). Falls Sie jetzt »G6000« eingeben, um das Programm zu starten, werden Sie sich sicherlich wundern, daß es dennoch läuft. Doch löschen Sie einmal das Programm in $4000 (mit »O4000 4100 AA«) und starten das Programm in $6000 noch einmal! Seltsam, nicht? Abhilfe schafft der nächste Befehl.

Variation

V (ANFADRalt ENDADRalt ANFADRneu ANFADR ENDADR) rechnet alle absoluten Adressen im Bereich von ANFADR bis ENDADR, die sich auf ANFADRalt bis ENDADRalt beziehen, auf ANFADRneu um. Kompliziert? Nicht, wenn Sie sich klarmachen, daß die ersten drei Adressen exakt den Eingaben beim »W«-Befehl entsprechen. Neu hinzu kommen nur die beiden Adressen für den Bereich, in dem die Änderung tatsächlich erfolgt.

Um unser mit »W« schon verschobenes Programm auch wieder lauffähig zu machen, geben Sie folgendes ein: V 4000 4020 6000 6000 600E. Damit werden alle Absolutadressen, die im Bereich von $6000 bis $600E – dahinter steht die Tabelle – liegen und sich bisher auf $4000 bis $4020 bezogen haben, auf den neuen Bereich umgerechnet. Probieren geht wie immer über kapieren.

Eine Zusammenfassung dieser beiden Befehle ermöglicht:

Convertieren

(Verschieben eines Programmes mit Adreßumrechnung.)

C (ANFADRalt ENDADRalt ANFADRneu ANFADRges ENDADRges) verschiebt das Programm von ANFADRalt bis ENDADRalt zur ANFADRneu und zwar mit Umrechnung der Adressen zwischen ANFADRges und ENDADRges

An unserem kleinen Testprogramm läßt sich wieder einmal demonstrieren, wie der Befehl eingesetzt wird. Laden Sie es also mit »L"SUPERTEST"« und schauen es mit »D 4000« an. Jetzt wollen wir an der Adresse $4008 einen 3-Byte-Befehl einfügen: C 4008 4020 400B 4000 4011 verschiebt das Programm von $4008 bis $4020 zur neuen Anfangsadresse $400B. Dabei werden im Bereich von $4000 bis $4011 (neue Endadresse des »aktiven« Programmes!) die Sprungadressen umgerechnet. Nun können Sie ab Adresse $4008 einen 3-Byte-Befehl einfügen, zum Beispiel STY 0286. Dazu geben Sie bitte ein:

```
A 4008
4008 STY 0286
F
```

Überzeugen Sie sich davon, daß SMON die Befehle korrekt umgerechnet hat, indem Sie unser Beispiel disassemblieren (D 4000) und anschließend mit G 4000 starten. Besitzer eines Farbmonitors werden in helle Begeisterung ausbrechen. Vorsicht ist geboten, wenn Tabellen oder Text vorhanden sind. SMON wird versuchen, diese als Befehle zu disassemblieren und gegebenenfalls umzurechnen. Dabei können unvorhersehbare Verfälschungen auftreten. Aus diesem Grunde ist im Beispiel die Endadresse des zu ändernden Bereiches auf $4011 und nicht etwa auf $4023 gelegt worden. Wenn Sie größere Programme zu verschieben haben, sollten Sie die Kommandos W und V anwenden beziehungsweise einen Assembler einsetzen (zum Beispiel Hypra-Ass), der es Ihnen gestattet, beliebige Einfügungen,

Verschiebungen und sonstige Änderungen vorzunehmen. Das C-Kommando eignet sich in erster Linie für kleinere Änderungen innerhalb eines Programms.

BASIC-DATA

B (Anfadr Endadr)

wandelt das Maschinenprogramm von ANFADR bis ENDADR-1 in Basic-DATA-Zeilen um.

B 4000 4020

Unser Testprogramm wird in DATA-Werte umgerechnet und dann mit Zeilennummer 32000 beginnend im Basic-Speicher abgelegt. Ein im Speicher befindliches Basic-Programm (zum Beispiel ein Basic-Lader) mit kleineren Zeilennummern kann dann diese DATA-Zeilen benutzen.

Wenn Sie das Testprogramm wie oben beschrieben umgewandelt haben, überzeugen Sie sich mit »LIST« von der Ausführung. Dann können Sie folgendes eingeben:

10 FOR I=16384 TO 16415 : READ D :POKE I,D : NEXT

In Verbindung mit den oben erzeugten DATA-Zeilen (und RUN!) hätten Sie wieder das ursprüngliche Maschinenprogramm im Speicher. Falls Sie dieses Beispiel durchführen wollen, denken Sie bitte daran, daß Sie nach Erstellung der DATAs das Originalprogramm zum Beispiel mit OCCUPY (O 4000 4020 AA) überschreiben, damit Sie die richtige Ausführung überprüfen können. Der BRK-Befehl am Ende des Testprogramms bewirkt einen Sprung zum SMON zurück. Wollen Sie ein Maschinenprogramm von Basic aus starten und auch wieder dorthin zurückgelangen, muß der letzte Befehl ein RTS sein. Probieren Sie es aus, indem Sie das Basic-Programm um 20 SYS 16384 erweitern.

KONTROLLE

K (Anfadr Endadr)

listet die ASCII-Zeichen im gewünschten Bereich. Es werden jeweils 32 Zeichen pro Zeile ausgegeben, so daß man sich einen schnellen Überblick über Texte oder Tabellen verschaffen kann.

Beispiel: K 4000 listet die ersten 32 Zeichen unseres Programms. Die weitere Ausgabe ist genau wie beim Disassemblieren durch Druck auf SPACE oder RETURN möglich. Auch hier können Sie wie bei den anderen Bildschirm-Ausgabebefehlen Änderungen durch einfaches Überschreiben vornehmen (natürlich nicht im ROM und nur mit ASCII-Zeichen!).

Als Beispiel wollen wir einmal im Basic »herumpfuschen«. Das geht natürlich nicht so ohne weiteres, weil das Basic im ROM steht und damit nicht verändert werden kann. Tippen Sie bitte folgendes ein:

W A000 C000 A000

Auf den ersten Blick eine unsinnige Anweisung; der Speicher soll von A000 bis C000 nach A000 verschoben werden. Dieser Befehl entspricht exakt der Basic-Schleife

FOR I = 40960 TO 49152 : POKE I, PEEK (I) : NEXT

Nun ist es aber so, daß beim PEEK das ROM gelesen, beim POKE aber ins darunterliegende RAM geschrieben wird. Wir erreichen also, daß das Basic ins RAM kopiert wird. Jetzt müssen wir dafür sorgen, daß das Betriebssystem sein Basic aus dem RAM und nicht aus dem ROM holt. Zuständig dafür ist die Speicherstelle 0001. Geben Sie bitte »M 0001« ein und überschreiben Sie die »37« mit »36«.

Es passiert gar nichts. Jetzt tritt unser K-Kommando in Aktion. Geben Sie ein: K A100 A360

Was Sie sehen, sind die Basic-Befehlswörter und -Meldungen. Schalten Sie mit SHIFT/CBM auf Kleinschrift, dann erkennen Sie, daß der jeweils letzte Buchstabe eines Befehlswortes groß geschrieben ist (Endekennung). Jetzt ändern Sie durch Überschreiben das »LIST« (A100) in »LUST« und »ERROR« (A360) in »FAELER«. (Bei »FAELER« müssen Sie ein Zeichen vor »ERROR« beginnen, sonst paßt es nicht.) Verlassen Sie jetzt SMON mit »X« und geben Sie danach ein:

POKE 1,54

SMON schaltet nämlich beim »X«-Befehl immer auf das Basic-ROM zurück, daher müssen wir wieder auf unser geändertes Basic umschalten. Schreiben Sie nun einen Basic-Dreizeiler und versuchen Sie, diesen zu LISTen. Ergebnis? Versuchen Sie es jetzt einmal mit »LUST«. Ihrer weiteren Phantasie sind keine Grenzen mehr gesetzt...

Wie oben angesprochen stellt SMON eine Reihe verschiedener Suchroutinen zur Verfügung, die im folgenden an vielen Beispielen beschrieben werden. Alle diese Befehle bestehen aus zwei Zeichen und beginnen mit dem Buchstaben »F«.

FIND

F (HEX-WERT(e), Anfadr Endadr)

sucht nach einzelnen HEX-Werten innerhalb eines bestimmten Bereichs. Das zweite Zeichen (hinter F) ist hier ein Leerzeichen und darf nicht weggelassen werden! Die Bereichsangabe kann wie bei allen folgenden Befehlen entfallen, dann wird der gesamte Speicher durchsucht.

Beispiel: Wir suchen alle Befehle LDY #01, also die Werte A0 01 im Bereich von $2000 bis $6000.

F A0 01, 2000 6000 (die Leerzeichen zwischen den Hex-Bytes dürfen nicht weggelassen werden!). Es erscheinen alle Speicherstellen, die die gesuchten Bytes enthalten, also zum Beispiel 4000.

FA (Adresse, Anfadr Endadr)

sucht alle Befehle, die eine bestimmte Adresse als Operanden haben (absolut). Die Adresse braucht nicht vollständig angegeben zu werden, es kann das Jokerzeichen »*« benutzt werden.

1. Beispiel: Wir suchen alle JSR FFD2-Befehle im Bereich $2000 bis $6000.

FAFFD2,2000 6000

Es erscheinen alle Befehle disassembliert, die FFD2 im Operanden enthalten (also auch LDA FFD2 oder STA FFD2,Y...).

2. Beispiel: Wir suchen alle Befehle, die auf den Grafikbereich ($D000 bis $DFFF) zugreifen.

FAD***,2000 6000

Der Joker kann aber auch zum Beispiel zur Suche im Bereich $D000 bis $D0FF dienen: FAD0**,2000 6000

FR (Adresse, Anfadr Endadr)

sucht nach relativen Sprungzielen. Anders als bei absoluten Sprüngen (JMP, JSR) benutzen die Branch-Befehle eine relative Adressierung, also zum Beispiel »Verzweige 10 vor« oder »37 zurück«. Solche Sprünge lassen sich mit dem FA-Kommando nicht finden. Hier wird »FR« eingesetzt.

Beispiel: Gesucht werden alle Branch-Befehle, die die Adresse $4002 anspringen.

FR4002,2000 6000

Natürlich können solche Befehle nur höchstens 128 Byte vom Sprungziel entfernt sein. Die Bereichsangabe ist hier also viel zu groß gewählt (SMON stört dies allerdings nicht). Der Einsatz des Jokers ist hier ebenfalls wie oben beschrieben möglich.

FT (Anfadr Endadr)

sucht Tabellen im angegebenen Bereich. SMON behandelt dabei alles, was sich nicht disassemblieren läßt, als Tabelle. Beispiel: Wir suchen Tabellen oder Text im Bereich $2000 bis $6000.

FT 2000 6000

FZ (Adr, Anfadr Endadr)
sucht alle Befehle, die Zeropage-Adressen haben.

1. Beispiel: FZC5,2000 6000 findet alle Befehle, die C5 adressieren, also zum Beispiel BIT $C5, LDA (C5), Y etc.

2. Beispiel: FZF*,2000 6000 findet alle Befehle, die den Bereich zwischen $F0 und $FF adressieren.

3. Beispiel: FZ**,2000 6000 findet sämtliche Befehle mit Zeropage-Adressierung.

FI (Operand, Anfadr Endadr)
sucht alle Befehle mit unmittelbarer Adressierung (immediate).

Beispiel: Gesucht werden Befehle, die zum Beispiel das Y-Register mit 01 laden. FI01,2000 6000 findet LDY #01 in Adresse $4000.

Sie sehen, SMON bietet eine Fülle von verschiedensten FIND-Routinen, mit denen alles gesucht und auch gefunden (!) werden kann.

= 4000 6000
vergleicht den Speicherinhalt ab $4000 mit dem ab $6000. Das erste nicht übereinstimmende Byte wird angezeigt und der Vergleich wird abgebrochen.

Wenn Sie also ein Maschinenprogramm geschrieben und überarbeitet haben und Sie wissen nicht mehr genau, worin eigentlich der Unterschied zwischen der 76. und der 77. Version besteht, gehen Sie so vor: Laden Sie zuerst Version 76 und verschieben Sie diese mit dem »W«-Befehl in einen freien Speicherbereich. Laden Sie dann Version 77 und führen Sie den »=«-Befehl durch. Sofort finden Sie den Unterschied und können mit der Arbeit an Version 78 beginnen ...

Wir wollen uns bei der Beschreibung der Trace-Befehle auf Anwendungsbeispiele konzentrieren. Zum Aufbau der Routine sei nur so viel gesagt: Gesteuert wird sie mit Hilfe des Prozessor-Interrupts, weil nur damit ein Eingriff ins laufende Maschinenprogramm möglich ist. Während des Trace-Ablaufs wird deswegen der Bildschirm kurzfristig aus- und eingeschaltet, weil alle anderen Interruptanforderungen wie zum Beispiel durch den Video-Chip, verhindert werden müssen. Da die Befehle eines Programms nicht nur angezeigt, sondern auch wirklich ausgeführt werden, ist der »SEI«-Befehl mit großer Vorsicht zu verwenden. Doch dazu später mehr. Wir wollen ein neues, besser geeignetes Beispiel verwenden als bisher. Tippen Sie also das folgende Miniprogramm mit dem Assembler ein (A 4000):

```
4000  LDA  #30    lade den Akku mit (ASCII-) 0
4002  JSR  FFD2   gib Akku auf dem Bildschirm aus
4005  CLC
4006  ADC  #01    erhöhe Akku um 1
4008  CMP  #39    vergleiche Akku mit (ASCII-) 9
400A  BCC  4002   springe, wenn Akku kleiner, zurück
400C  BRK         springe in SMON zurück
```

Starten Sie das Programm mit »G 4000«. Es muß die Zahlen von 0 bis 8 auf den Bildschirm schreiben.

◆ Trace-Stop
TS (Startadresse Stoppadresse)
Starten Sie nun unser Programm mit TS 4000 4009. Die ersten Befehle werden ausgeführt (die Null ausgegeben, der Akku erhöht etc.), dann stoppt das Programm bei Adresse $4009 und springt in die Registeranzeige.

Genau genommen ist »TS« gar kein Trace-Befehl, das Programm läuft nämlich bis zur gewählten Stoppadresse in Echtzeit durch. Dort angekommen, können Sie die Register prüfen und gegebenenfalls durch Überschreiben ändern. Mit »G«, »TW« oder »TB« (wird später erklärt) ohne weitere Adresseneingaben können Sie dann im Programmlauf fortfahren. SMON merkt sich nämlich, wo er stehengeblieben ist und arbeitet ab dieser Adresse weiter, wenn Sie nicht eine neue angeben.

Sinnvoll ist dieser Befehl immer dann, wenn in einem längeren Programm nur bestimmte Teile »getraced« werden sollen, der Anfang aber durchlaufen werden muß, um Variable zu setzen oder Benutzereingaben zu erfragen. Auch wenn man nicht ganz sicher ist, ob eine bestimmte Passage überhaupt jemals durchlaufen wird, kann man das mit »TS« überprüfen.

Zwei Einschränkungen gibt es allerdings wegen der Arbeitsweise dieses Befehls: SMON setzt im Programm an die Stoppadresse einen BRK-Befehl und merkt sich, welcher Befehl dort stand, um ihn wieder zurückzuschreiben. Deshalb funktioniert »TS« nur im RAM, nicht aber zum Beispiel im Basic oder im Betriebssystem. Auch darf die Speicherstelle, in der sich SMON den ausgetauschten Befehl merkt ($02BC) vom Programm nicht verändert werden, sonst ist eine korrekte Reparatur nicht mehr möglich.

Der wohl am häufigsten und vielseitigsten eingesetzte Trace-Befehl ist sicherlich »TW«.

◆ Trace Walk
TW (Startadresse)
Starten Sie unser Beispiel jetzt mit TW 4000

Der erste Befehl (LDA #30 in Adresse $4000) wird ausgeführt, SMON stoppt und zeigt dann die Inhalte aller Register in der gleichen Reihenfolge wie beim »R«-Kommando sowie den nächsten Befehl an. Im Akku steht jetzt 30, der Programmzähler zeigt auf $4002. Jetzt drücken Sie eine Taste. Der nächste Befehl (JSR FFD2) wird ausgeführt, der Programmzähler zeigt auf $FFD2. Achten Sie auf den Stackpointer: Sein Inhalt hat sich um 2 vermindert, weil der Prozessor auf dem Stack die Adresse abgelegt hat, an die er nach Beendigung der Subroutine zurückspringen soll. Der nächste angezeigte Befehl ist ein indirekter Sprung über $0326. Mit dem nächsten Tastendruck wird er durchgeführt.

Und so geht es munter weiter. Verzweifeln Sie nicht, wenn Sie auch nach den nächsten zehn Tastendrücken immer noch irgendwo im Betriebssystem »herumtracen« und von unserem Beispielprogramm weit und breit nichts mehr zu sehen ist. Ausnahmsweise ist unser Liebling einmal nicht im »Land der Träume« verschwunden, sondern tut, was er soll: Er arbeitet brav einen Befehl nach dem anderen ab, was zur Routine $FFD2 gehört, und das ist reichlich viel. Also bewegen Sie Ihre Finger, Sie haben's ja nicht anders gewollt. Irgendwann einmal, nach mehreren hundert gedrückten Tasten, befinden Sie sich plötzlich wieder in der Registeranzeige von SMON. Das Programm ist beendet. Nun werden Sie enttäuscht fragen, was man wohl mit einem Trace-Modus anfangen soll, der schon bei kleinsten Beispielprogrammen ein völlig undurchschaubares Chaos erzeugt? Nur Geduld, die Rettung naht in Gestalt der Taste »J«.

Falls ihre Hand noch nicht in Gips liegt, starten Sie das Ganze nochmal von vorn mit »TW 4000«. Diesmal drücken Sie aber jedesmal, wenn als nächster Befehl »JSR FFD2« angezeigt wird, auf »J«. Der Effekt ist, daß die gesamte Subroutine auf einen Schlag abgearbeitet wird und Sie sofort wieder auf dem nächsten Befehl unseres Beispiels landen. Daß wir nicht gemogelt und die Befehle von »JSR FFD2« einfach unterschlagen haben, sehen Sie daran, daß der Akku tatsächlich auf dem Bildschirm ausgegeben worden ist (rechts neben FFD2). Jetzt können Sie unser Beispiel in aller Ruhe bis zum Ende durchgehen und verfolgen, wie der Akku erhöht wird, wie der Vergleich das Statusregister beeinflußt und wie entsprechend der Rücksprung in die Schleife erfolgt.

Sie dürfen die »J«-Taste auch dann benutzen, wenn Sie schon mitten in der Subroutine sind. Aber hierbei ist äußerste Vorsicht geboten: Die Rücksprungadresse muß unbedingt oben auf dem Stack liegen, wenn Sie »J« drücken. Hat nämlich der Prozessor Werte auf dem Stack abgelegt (mit PHA oder PHP), dann erfolgt der Sprung irgendwo hin, nur nicht zurück ins Programm. Achten Sie deshalb genau auf die Anzeige des Stackpointers. Wenn dessen Wert genau so

groß ist wie bei Beginn der Subroutine, kann nichts passieren. Sonst hilft nur noch der Reset-Taster, den Sie ja inzwischen hoffentlich eingebaut haben, oder eine ruhige Hand, die die Büroklammer an Pin 1 und 3 des User-Ports hält (Kostenpunkt der Reparatur bei Abrutschen liegt bei zirka 100 Mark ...).

»TW« bricht automatisch mit der Registeranzeige ab, wenn im Programm ein »BRK«-Befehl auftaucht. Wenn Ihnen das zu lange dauert oder Sie zwischendurch ein Register ändern möchten, können Sie den Trace-Modus jederzeit mit der Stopp-Taste verlassen. Anschließend können Sie wie bei »TS« beschrieben fortfahren.

Im Gegensatz zu »TS« können Sie mit »TW« auch im ROM herumstöbern; Sie haben es ja bei der Subroutine $FFD2 bereits getan. Einzige Einschränkung beim »TW«-Befehl: Ihr Programm darf keinen »SEI« enthalten, da dieser den Interrupt und damit auch den Trace-Modus lahmlegt. Verlassen Sie in diesem Falle »TW« mit STOP und starten erneut hinter dem »SEI«-Befehl. Allerdings müssen Sie in Kauf nehmen, daß das Programm normalerweise nicht mehr korrekt arbeitet.

Das nächste Programm soll als weiteres Beispiel für den TW-Modus dienen. Geben Sie es folgendermaßen ein:

5000	LDA	#00	lädt den Akku mit »0«
5002	TAX		überträgt den Akku ins X-Register
5003	.0C		ein mysteriöses Byte
5004	LDA	#04	lädt den Akku mit »4«
5006	TAY		überträgt den Akku ins Y-Register
5007	BRK		springt in SMON

Wenn wir dieses kleine Programm abarbeiten, müßte das X-Register auf »0« stehen, während Akku und Y-Register mit »4« geladen sind. Starten wir also das Programm mit »G 5000« und schauen uns die Register an.

Seltsamerweise enthalten alle Register eine »0«. Vorsichtig, wie wir sind, überschreiben wir die drei Register mit »FF«, um die Veränderung deutlich kontrollieren zu können.

Dann starten wir mit »G 5000« ein zweites Mal. Gegen alle Gesetze der Vernunft erscheint wieder das »falsche« Ergebnis – alle drei Register sind »0«. Hier soll uns jetzt der TW-Modus weiterhelfen, indem er uns zeigt, was in Wirklichkeit passiert.

Geben wir »TW 5000« ein. Der erste Befehl (LDA #00) ist durchgeführt, im Akku erscheint die Null. Jetzt steht der Programmzähler auf dem folgenden Befehl »5002 TAX«. Nach Drücken einer Taste wird dieser Befehl ausgeführt und es erscheint die Null im X-Register. Beim folgenden Befehl müssen wir feststellen, daß der Disassembler nicht in der Lage ist, ihn zu interpretieren – er gibt drei Sternchen aus. Hierbei handelt es sich um unser Byte »0C«.

Wieder ein Tastendruck; und dann erkennen wir, daß etwas Merkwürdiges passiert ist. Der Prozessor hat augenscheinlich den nächsten Befehl (LDA #04) übersprungen und steht schon auf dem folgenden »TAY«. So also wird unser Programm abgearbeitet. Damit ist auch das »falsche« Ergebnis erklärt. Bleibt nur noch die Frage nach dem Grund für dieses seltsame Verhalten. Und der ist sicherlich in dem mysteriösen Byte »0C« zu suchen. Hierbei handelt es sich um einen der »inoffiziellen« Opcodes, die aufgrund der Prozessorarchitektur vorhanden sind und in manchen Programmen ihr Unwesen treiben – wie wir zu unserem Leidwesen erfahren mußten. Das Byte »0C« wirkt wie ein »NOP«, der eine Länge von 3 Byte hat. Deshalb wird der folgende 2-Byte-Befehl (LDA #04) verschluckt.

Es gibt noch einiges zu entdecken am 6502 und 6510 – TW macht's möglich.

Häufig ist es nicht sinnvoll, ein Programm von Anfang an im TW-Modus laufenzulassen. Zum anderen sind gerade Schleifen, die per Hand mit »TW« durchlaufen werden müssen, eine ermüdende Angelegenheit. Hier bietet SMON neben dem bereits beschriebenen »TS« eine weitere Trace-Möglichkeit an:

◆ **Trace Break**

TB (Adresse Anzahl der Durchläufe)

◆ **Trace Quick**

TQ (Adresse)

Geben Sie als Beispiel folgendes Programm ein:

6000	LDY	#00	Y als Zähler auf »0«
6002	LDA	600E,Y	Werte von $600E ff. sollen geladen werden
6005	JSR	FFD2	Ausgabe der Zeichen auf dem Bildschirm
6008	INY		der Zähler wird erhöht
6009	CPY	#0E	Zähler schon »14«?
600B	BCC	6000	wenn nein, dann nächsten Wert holen
601D	BRK		

Bei $600E soll nun ein Text stehen, den das Programm ausgibt. Die einfachste Art, mit SMON Texte in den Speicher zu schreiben, besteht im »K«-Befehl. Geben Sie

K 600E

ein (danach natürlich Return) und drücken Sie die STOP-Taste. Fahren Sie mit dem Cursor an das erste ausgegebene Zeichen (vermutlich ein Punkt) und schreiben Sie – ohne Anführungszeichen:

»FEHLER BEHOBEN«

Drücken Sie dann Return, um die Zeile an den Rechner zu übergeben. Wenn Sie das Programm starten, werden Sie wieder einmal Gelegenheit haben, sich in Ruhe etwas zu trinken zu holen (Prost!), denn das Programm enthält einen dummen Fehler und beschäftigt den Computer für eine lange, lange Zeit. Genauer gesagt, bis Sie ihn mit Reset (zum Beispiel durch RUN/STOP-RESTORE) erlösen.

Nun soll SMON helfen, diesen Fehler zu lokalisieren. Setzen Sie zuerst einmal einen Breakpoint bei $6002 und begrenzen die Durchläufe auf die maximale Anzahl:

TB 6002 0E

und starten mit

TQ 6000

den Quicktrace bei $6000. Das Programm läuft so lange, bis zum 14. Mal die Adresse $6002 erreicht wird und springt dann in den TW-Modus. Wenn Sie sich jetzt die Registerinhalte genau anschauen, müßte ihnen der Fehler geradezu ins Auge springen. Wie groß sollte denn das Y-Register sein? Welchen Wert sollte der Akku haben? NA?!

Das »Gedächtnis« von SMON

Wenn Sie Programme mit SMON untersuchen oder verändern wollen, müssen Sie noch wissen, welche Speicherstellen SMON verwendet. Es soll ja Monitorprogramme geben, die die Basic-Zeiger als Arbeitsspeicher benutzen, so daß ein Basic-Programm nach dem Rücksprung aus dem Monitor gelöscht ist. SMON tut so etwas nicht. Aber natürlich braucht er auch Speicherstellen, um sich Werte merken zu können. Damit Sie Konflikten von Anfang an aus dem Wege gehen können, sind die wichtigsten hier dargestellt.

In der Zeropage belegt SMON den Bereich von $00A4 bis $00B6. Dort stehen Systemvariable für die Kassettenspeicherung und die RS232-Schnittstelle. Diese werden nur während des Betriebs der Kassette oder von RS232 gebraucht, sind ansonsten aber frei. Außerdem werden die Speicherstellen $00FB bis $00FF benutzt, die sowieso zur freien Verfügung des Anwenders vorgesehen sind. Alle anderen Zeiger in der Zeropage, also insbesondere die Speicherverwaltung für Basic, bleiben unbeeinflußt.

Als weiteren Arbeitsspeicher benutzt SMON den Bereich von $02A8 bis $02C0. Auch dieser Bereich wird vom Betriebssystem nicht benutzt, so daß keine Konflikte entstehen dürften. Beim Assemblieren wird zusätzlich noch der Kassettenpuffer als Speicher für die Label benötigt. Dieser bleibt ansonsten aber auch unverändert; das ist wichtig, wenn Maschinenroutinen dort abgelegt werden sollen.

Alles in allem ist SMON also recht verträglich.

SMON verschieben? - Mit SMON!

Eine Reihe von Anfragen hat uns erreicht, ob man SMON nicht mit Hilfe des »W«-, »V«- oder »C«-Kommandos verschieben könne. Alle Versuche in dieser Richtung seien fehlgeschlagen. Einige Leser meinten auch, in der V-Routine müsse ein Fehler stecken. Diesmal sind wir jedoch völlig schuldlos; es gibt nämlich einige Befehle in SMON, die keine Sprungadressen sind und sich trotzdem auf den Bereich ($C000-) beziehen, in dem SMON steht.

Dazu gehören in erster Linie die oben erwähnten Einsprungadressen, deren High-Byte natürlich geändert werden muß, wenn SMON in einem anderen Speicherbereich laufen soll. Es gibt aber auch Befehle, die eine Adresse im Programm in einem Vektor ablegen müssen. Disassemblieren Sie einmal den Anfang von SMON mit »D C000 C00B«. Sie erhalten

```
LDA   #14    Low-Byte der BREAK-Routine von SMON
STA   0316   im Break-Vektor speichern
LDA   #C2    High-Byte (!) siehe oben
STA   0317   siehe oben
BRK
```

Damit wird der Break-Vektor des Betriebssystems auf den SMON gesetzt und mit dem anschließenden — und jedem weiteren BRK-Befehl — springt das Programm in SMONs BREAK-Routine. Wenn SMON in einem anderen Bereich als $C000 laufen soll, dann müssen diese Befehle geändert werden.

Heraussuchen kann man sie mit »FIC*,C000 D000«. Sie wissen doch noch, was diese Anweisung bedeutet: Suche mir alle Befehle, die ein Register unmittelbar mit einem Wert laden, der mit $C beginnt. Aber Vorsicht! Nicht alles, was da angezeigt wird, muß auch geändert werden! Um Ihnen weitere Stunden sinnlosen Herumbrütens zu ersparen, wollen wir als Beispiel zeigen, wie man SMON in den Bereich $9000 bis $A000 verlegen kann. Natürlich geht das im Prinzip für jeden anderen Bereich genauso; wir selbst haben insgesamt fünf SMON-Versionen für fünf verschiedene Speicherbereiche, von denen eine immer paßt.

1. Wir verschieben zuerst das ganze Programm ohne Umrechnen in den neuen Bereich:
W C000 CFFA 9000

2. Nun lassen wir alle absoluten (3-Byte-)Befehle umrechnen. Die Tabellen am Anfang von SMON bleiben verschont:
V C000 CFFA 9000 920B 9FD2

3. Als nächstes ändern wir die High-Bytes der Befehlsadresse. Geben Sie
»M 902B 906B«
ein und ändern Sie in jedem zweiten Byte das »C« durch Überschreiben in »9«. Vergessen Sie nicht, am Ende jeder Zeile »RETURN« zu drücken, damit Ihre Änderung auch übernommen wird.

4. Nun sind die Befehle mit Immediate-Adressierung an der Reihe. Sie müssen so geändert werden, daß sie sich auf den neuen Bereich $9... beziehen. Suchen Sie sie mit
FIC*,9000 9FFA
heraus. Sie erhalten

```
9005  LDA  #C2  ändern
9124  CPX  #C0  nicht ändern
9386  LDY  #C0  ändern
9441  CMP  #C0  nicht ändern
987F  LDX  #C3  nicht ändern
988D  LDX  #C1  nicht ändern
9992  LDA  #C1  nicht ändern
9C2C  LDA  #CC  ändern
9C5B  LDA  #C2  ändern
9CF4  LDA  #CC  ändern
9DA1  LDX  #CC  ändern
9E03  LDA  #CC  ändern
9E6C  CMP  #C0  nicht ändern
9F71  LDY  #CF  ändern
```

Sie sehen, es gibt keine Regel, welche Befehle zu ändern sind und welche nicht. Aus diesem Grunde müssen Sie diese Änderungen »von Hand« vornehmen.

5. Die Adressen im Diskmonitor müssen ebenfalls umgestellt werden. Dazu geben Sie bitte ein:
M 9FD8 9FE4
und ändern Sie jedes zweite Byte wie unter Punkt 3 beschrieben.

Vergessen Sie bitte auf keinen Fall, Ihre neue(n) Version(en) unter neuem Namen zu speichern. Sie lassen sich dann mit LOAD "Name",8,1 von Diskette laden und mit dem entsprechenden SYS (zum Beispiel 36864 bei SMON $9000) starten. Denken Sie auch daran, nach dem Laden und vor dem SYS ein NEW einzugeben, sonst beschwert sich der B-Befehl mit einem OUT OF MEMORY ERROR.

Probieren Sie nun alle Befehle durch. Sie müssen genauso arbeiten wie bisher. Vor allem können Sie jetzt auch Programme wie »DOS 5.1« oder »Turbo Tape« untersuchen, die im $C000-Bereich stehen. Achten Sie aber, wenn Sie »SMON $9000« von Basic aus benutzen, darauf, daß das Basic ihn nicht überschreibt. String-Variable werden nämlich von $A000 nach unten hin aufgebaut und bis $9E09 ist nicht viel Platz. Schützen Sie im Zweifelsfalle den Bereich, indem Sie nach dem Laden des SMON $9000 eingeben:

NEW : POKE 56,144 : POKE 55,0

Damit ist SMON vor Überschreiben geschützt. Das ist natürlich bei dem SMON $C000 nicht nötig, weil Basic in diesen Bereich nicht hineinkommt.

Die Befehle des Disk-Monitors

Da das Arbeiten mit dem Disk-Monitor besondere Aufmerksamkeit verlangt (nach Murphys Gesetzen führen Fehleingaben in der Regel zu unlesbaren Disketten), wird er mit einem eigenen Kommando eingeschaltet. Leider waren alle halbwegs sinnvollen Buchstaben (»D« wie Diskette oder »F« wie Floppy) schon vergeben, deshalb haben wir uns für ein schlichtes »Z« wie Zuversicht entschieden.

-Z schaltet den Disk-Monitor ein

Die Rahmenfarbe ändert sich auf Gelb, der gewohnte ».« am Anfang einer Zeile ändert sich in »*«. Dies alles hat den Zweck, Ihnen deutlich zu machen, daß es jetzt ernst wird. Intern wird jetzt das Basic abgeschaltet, weil der Disk-Monitor einen 256 Byte großen Puffer benötigt. Dieser liegt von $BF00 bis $C000 im RAM unter dem Basic, weil er dort am wenigsten stören kann.

READ: R (Track Sektor)

Liest einen Block von der Diskette in den Computer. Track und Sektor müssen als Hexzahlen eingegeben werden. Die erste Zeile des Blocks wird ausgegeben. Da wir dazu normale SMON-Routinen verwenden, steht als Speicheradresse $BF00. Das »BF« können Sie vorerst ignorieren. Die weitere Ausgabe des Hexdump erfolgt anders als gewohnt mit der Taste »SHIFT«. STOP bricht die Ausgabe ab. Sie können die Hex-Bytes überschreiben und damit ändern. Eine dauerhafte

Änderung erfolgt aber erst beim Zurückschreiben auf die Diskette (siehe Befehl »W«). Geben Sie nur »R« ohne Track und Sektor ein, wird der logisch (!) nächste Block eingelesen.

MEMORY-DUMP: M

Zeigt den gerade im Puffer befindlichen Block nochmals auf dem Bildschirm an.

Genau wie beim R-Befehl können Sie die Ausgabe mit »SHIFT« und »STOP« steuern und Änderungen vornehmen.

WRITE: W (Track Sektor)

Schreibt einen Block aus dem Puffer auf die Diskette zurück. Ähnlich wie bei »R« kann die Angabe von Track und Sektor entfallen. Es wird dann der Track und Sektor des letzten R-Befehls benutzt. Das ist in fast allen Fällen auch der richtige.

ERROR: @

Liest den Fehlerkanal aus, gibt ihn aber nur aus, wenn wirklich ein Fehler vorhanden war. (»00, OK, 00, 00« wird unterdrückt.)

EXIT: X

Verläßt den Disk-Monitor und springt in den SMON zurück. Dabei wird die Rahmenfarbe auf Blau zurückgeschaltet und es erscheint wieder der ».« am Anfang der Zeile. Das Basic wird wieder eingeschaltet. Wollen Sie nun mit SMON-Kommandos auf den Puffer zugreifen, müssen Sie Basic wieder abschalten ($36 in Speicherstelle $0001).

Die folgenden Beispiele sollen Ihnen die Arbeit mit dem Disk-Monitor verdeutlichen.

Achtung! Benutzen Sie unbedingt zum Üben eine Diskette, die Sie nicht mehr brauchen!

Weder wir noch der Verlag haften dafür, wenn Ihr Lieblingsprogramm oder die mühsam erstellte Adreßdatei unwiederbringlich dahin sind. Daß das sehr sehr schnell gehen kann, wissen wir aus eigener Erfahrung ...

Am besten machen Sie von einer Ihrer Diskette eine Kopie, die Sie zum Üben benutzen können.

Reparatur eines gelöschten Files

Sicher ist Ihnen das auch schon passiert: Sie wollen Ihr Programm mit Namen »Schrott« löschen, geben als Abkürzung »S:S*« ein und merken in dem Moment, in dem Sie »RETURN« drücken, daß auf der Diskette auch alle Versionen von »SMON« waren, außerdem auch noch »Springvogel«, »Soccer« etc. Verzweifeln müssen Sie nur, wenn auch diese letzte SMON-Version mit dem Disk-Monitor dabei war. Ansonsten behalten Sie die Ruhe und verfahren Sie wie im folgenden beschrieben.

Laden Sie also jetzt SMON, legen Sie Ihre »Übungsdiskette« (!) ins Laufwerk und löschen Sie eins der ersten Programme mit dem üblichen Scratch-Kommando. Nun starten Sie SMON und drücken »Z«. Der Bildschirm ändert seine Farbe wie beschrieben und am Anfang der Zeile erscheint der »*«. Jetzt geben Sie ein:

R 12 00

Auf dem Bildschirm erscheint die erste Zeile der BAM, die bei jeder Diskette auf Track 18, Sektor 0 abgelegt ist. Die ersten beiden Bytes enthalten »12 01« und geben damit den logisch nächsten Block an. In diesem Falle wäre das der erste Block des Directory. Wenn Sie mit »SHIFT« die Bildschirmausgabe fortsetzen, erkennen Sie etwa in der Mitte den Diskettennamen. Lassen Sie die Ausgabe durchlaufen, bis wieder der »*« erscheint. Nun geben Sie »R« ohne weitere Angaben ein. Damit erhalten Sie den Koppel-Block, also Track 18, Sektor 1, den ersten Directory-Block. (Natürlich hätten Sie auch gleich »R 12 01« eintippen können, aber wir wollen ja zeigen, wie die Befehle funktionieren.)

In diesem Block stehen die ersten acht Programme Ihrer Übungsdiskette, auch der Name des gelöschten ist dabei.

Trotzdem ist dieses Programm tatsächlich gelöscht und erscheint nicht mehr, wenn Sie sich das Directory anzeigen lassen. Vergleichen Sie den Eintrag des gelöschten Programms mit den anderen, fällt auf, daß 3 Byte vor Beginn des Namens bei allen anderen »82« steht (sofern es sich um Programmfiles handelt), bei dem gelöschten aber »00«. Die Reparatur ist nun denkbar einfach: Sie brauchen lediglich die »00« mit »82« zu überschreiben. Einen Haken hat die Sache allerdings noch. Beim SCRATCHEN sind die vom Programm belegten Blöcke in der BAM als frei gekennzeichnet worden und jeder neue Eintrag würde das als gelöscht gekennzeichnete File endgültig überschreiben. Um das zu verhindern, müssen Sie nach erfolgter Reparatur die Diskette validieren (von Basic aus mit Kommando: OPEN 1, 8, 15, "V"). Dabei wird die BAM neu erzeugt und korrigiert.

Schützen eines Files

Da wir gerade dabei sind, wollen wir unser repariertes gelöschtes File gleich ein für allemal gegen Löschen schützen. Diese Möglichkeit des Diskettenoperationssystems (DOS) ist zwar nicht im Handbuch beschrieben, funktioniert aber trotzdem ausgezeichnet. Laden Sie dazu nochmals die erste Seite des Directory mit

R 12 01

und ändern Sie die »82« vor dem Fileeintrag in »C2«. Geben Sie »W« ein, um die Änderung auf Diskette zu schreiben. Verlassen Sie nun SMON mit »X« und lassen Sie sich ein Directory anzeigen. Das geschützte File ist mit einem »>« gekennzeichnet. Versuchen Sie nun, dieses Programm mit dem Scratch-Kommando zu löschen. Es geht nicht! Zum »Entriegeln« brauchen Sie nur das »C2« wieder in »82« zu ändern. Der »>« im Directory verschwindet und das File ist nicht mehr geschützt.

Schützen einer Diskette

Wollen Sie eine ganze Diskette vor versehentlichem Löschen oder Formatieren schützen, gibt es die Möglichkeit, die Löschschutzkerbe abzukleben. Es geht jedoch auch anders.

Achtung! Die im folgenden beschriebene Prozedur läßt sich nicht ohne weiteres rückgängig machen, auch nicht mit dem Disk-Monitor!

Nehmen Sie also eine Diskette, die Sie anschließend »hart formatieren« können (also mit Eingabe einer ID). Starten Sie nun den Disk-Monitor und lesen Sie die BAM mit »R 12 00« ein. Das dritte Byte enthält »41«. Diese »41« ist ein Kennzeichen für das DOS der 1541- oder 4040-Floppy. Ändern Sie diese Byte durch Überschreiben in »45« und speichern Sie die Änderung mit »W« auf die Diskette zurück. Verlassen Sie nun SMON und versuchen Sie, etwas zu löschen. Ergebnis siehe oben. Versuchen Sie auch, die Diskette »weich«, also zum Beispiel mit OPEN 1,8,15,"N:TEST" zu formatieren.

Auch das ist jetzt nicht mehr möglich. Aber es kommt noch besser: Starten Sie noch einmal den Disk-Monitor und versuchen Sie, die Änderung durch Zurückschreiben der »41« an Stelle der »45« rückgängig zu machen. Auch das ist nicht mehr möglich, wir hatten Sie bereits gewarnt! Es bleibt lediglich die Möglichkeit, die Diskette »hart«, zum Beispiel mit OPEN 1,8,15,"N:TEST,TE" zu formatieren. Sollten Sie nun entgegen allen Warnungen doch Ihre Master-Diskette gegen Schreibzugriffe gesichert haben, verraten wir Ihnen ausnahmsweise, wie Sie den Eingriff trotzdem rückgängig machen können. Dazu überlisten wir das DOS des 1541-Laufwerkes, indem wir ihm vorgaukeln, es hätte eine Diskette im Normalformat vor sich. Wir verwenden den Memory-Write-Befehl, mit dem wir in die Speicherstelle 0101 (Zero-Page Adresse) des 1541-RAM einfach ein »A« schreiben. Der CHR$-Code des »A« ist 65, oder in hexadezimaler Schreibweise 41. Erinnern Sie sich? Dieser Wert stand ursprünglich im dritten Byte des Tracks 18, Sektor 0. Mit folgendem kleinen Programm umgehen wir einfach die DOS-Kennzeichnung und wir können die Diskette wieder normal beschreiben. Am sinnvollsten ist es, sofort den SMON zu starten, das

vorher in 45 abgeänderte Byte wieder in 41 zu verwandeln und abzuspeichern. Die Diskette kann dann wieder zum Lesen und Schreiben verwendet werden. Hier nun das kleine Programm:

```
10 OPEN 1,8,15
20 PRINT#1, "M-W"CHR$(1)CHR$(1)CHR$(1)CHR$(65)
30 CLOSE1
```

Ändern des Diskettennamens oder der ID

Wir haben bereits oben gesehen, daß in Spur 18, Sektor 0 einer Diskette etwa in der Mitte der Diskettenname gespeichert wird. Dieser Name kann durch einfaches Überschreiben geändert werden; er darf bekanntlich bis zu 16 Zeichen enthalten. Hat Ihr neuer Name weniger Buchstaben als der alte, müssen Sie die Lücken mit »A0« und nicht mit »20« als Leerzeichen ausfüllen. Dies gilt vor allem, wenn Sie mit dieser Methode Filenamen ändern wollen. Das geht natürlich im Prinzip genauso wie eben beschrieben. Hinter dem Diskettennamen ist in Spur 18, Sektor 0 die ID abgelegt. Sie wird beim Formatieren vor jeden Sektor in einen sogenannten Header geschrieben und dient dem DOS zur Identifikation der Diskette. Zusätzlich wird sie noch in der BAM gespeichert, damit sie beim Laden eines Directory mit angezeigt werden kann. Nun ist es grundsätzlich nicht möglich, die ID im Header eines Sektors ohne Formatieren zu ändern, wohl aber die Eintragung in der BAM und damit die ID, die im Directory angezeigt wird. Genau wie beim Namen ist dies durch einfaches Überschreiben in der BAM möglich.

Ändern eines Filetyps

Wenn Sie einmal versucht haben, ein sequentielles File, etwa eine Datei, mit LOAD zu laden, werden Sie gemerkt haben, daß dies nicht möglich ist. Das DOS behauptet einfach, ein solches File existiere nicht und der Computer meldet »FILE NOT FOUND«. Viele Spiele zum Beispiel legen die »Hall of Fame« oder Highscore-Liste als sequentielle Datei ab. Mit dem Disk-Monitor ist es nun aber möglich, den Filetyp im Directory zu verändern. Erinnern Sie sich an die »82«, die im Directory vor jedem Filenamen steht. Bei sequentiellen Files steht dort »81«. Was zu tun ist, werden Sie sich denken können. Na klar, die »81« wird in »82« geändert, und schon ist die Datei ohne weiteres ladbar, natürlich wieder erst nach dem Zurückschreiben mit »W«.

Sinnvoll ist dies natürlich nur von SMON aus (mit Eingabe einer Ladeadresse). Mit »M« oder »K« können Sie dann die Datei ansehen und natürlich auch ändern. Vergessen Sie nicht, die geänderte Datei nach dem Zurückschreiben wieder in ein sequentielles File zu verwandeln. Verblüffen Sie Ihre Freunde doch mal mit einem auf diese Weise »errungenen« High-Score. Die Anerkennung Ihrer Umwelt ist Ihnen sicher!

Ändern der Startadresse eines Programms

Wir haben uns bisher auf Manipulationen in der BAM oder im Directory beschränkt. Wollen wir in einem Programm selbst Änderungen vornehmen, müssen wir etwas tiefer in die »Geheimnisse der Floppy« eindringen. So ist es bisweilen interessant, die Startadresse eines Maschinenprogramms zu kennen oder zu ändern. Dazu gehen wir folgendermaßen vor: Zunächst suchen wir mit »R 12 01« und eventuell weiteren Folgesektoren (12 04, 12 07...) den Fileeintrag im Directory. Die beiden Byte hinter der »82« direkt vor dem Programmnamen geben an, auf welcher Spur und in welchem Sektor das Programm startet. Wenn dort zum Beispiel »0A 04« steht, beginnt das Programm auf Spur 10, Sektor 4. Lesen Sie nun diesen Block mit »R 0A 04« ein. Die ersten beiden Bytes dieses Blocks zeigen auf den nächsten Block des Programms, die beiden nächsten Bytes enthalten die Startadresse in der üblichen Low-High-Byte-Reihenfolge. Zum Ändern der Startadresse überschreiben Sie die Bytes mit der neuen und speichern den Block mit »W« auf die Diskette zurück.

Die Zusammenarbeit mit SMON

Mit all diesen Beispielen sind die Möglichkeiten des Disk-Monitors noch lange nicht erschöpft. Sie sollten Ihnen als Anregung für eigene Experimente dienen. Üben Sie aber unbedingt so lange, bis Sie alle Kommandos aus dem »FF« (oder dezimal 255) beherrschen. Sie ersparen sich damit unnötigen Ärger und durchweinte Nächte. Besonders interessant ist es, von SMON aus auf den Puffer zuzugreifen und die SMON-Befehle auf den Puffer anzuwenden. Erwähnen möchte ich nur die Möglichkeit, Programme für das DOS direkt zu assemblieren und in einem bestimmten Sektor ablegen zu können, die »Find«-Routinen oder das »K«-Kommando für Textänderungen. Da der Puffer im RAM unter dem Basic liegt, muß Basic in solchen Fällen abgeschaltet werden. Ändern Sie dazu mit dem »M«-Befehl in Speicherstelle 0001 die »37« in »36«.

Haben Sie die Arbeit mit SMON beendet, können Sie mit »Z« in den Disk-Monitor schalten und den Pufferbereich mit »W« (Spur, Sektor) abspeichern.

Die Ausgabe von Diskettenfehlern

Beim Arbeiten mit dem Disk-Monitor werden sämtliche Fehler vom Laufwerk direkt, auch ohne Eingabe von »@«, ausgegeben, zum Beispiel »ILLEGAL TRACK OR SECTOR«, wenn Sie mit »R« einen Block lesen wollen, den es gar nicht gibt. Einen Fehler hat das Programm allerdings, den wir nicht verschweigen wollen. Der letzte Block eines Files enthält als Koppeladresse »00 FF«. Da es einen solchen Block nicht geben kann, »weiß« das DOS, daß es am Ende angelangt ist. Versuchen Sie aber, den nächsten Block (Spur 0, Sektor 255!!) mit »R« zu lesen, erscheint als Fehlermeldung nicht, wie es sein müßte, »ILLEGAL BLOCK OR SECTOR«, sondern »SYNTAX ERROR«. Das ist zwar eigentlich unerheblich, sollte aber erwähnt werden. Der Fehler liegt in der Routine, die unsere Zahleneingaben in das richtige Diskettenformat wandelt. Es fehlte einfach der Platz im Programm für eine »korrekte« Umwandlung, wir mußten uns mit einer »Sparroutine« behelfen.

Abschließend noch ein SMON-Trick, den wir einem aufmerksamen Leser verdanken. Für eine Directory-Ausgabe fehlte der Platz im SMON. Es geht aber hilfsweise so: Laden Sie das Directory zum Beispiel mit

L "$" 8000

an einen freien Speicherplatz. Mit »M« oder »K« können Sie jetzt das Directory »lesen«. Damit sind alle wichtigen Funktionen für den Umgang mit der Diskette im SMON enthalten.

SMON lüftet Geheimnisse

Zwei Erweiterungen haben wir Ihnen zu Beginn angekündigt, die SMON noch leistungsfähiger machen sollen. Dabei handelt es sich einmal um eine Erweiterung des Disassemblers, mit dem nun auch die »illegalen« Opcodes des 6502 disassembliert werden, zum anderen um neue Funktionen beim Diskmonitor, mit denen Sie in den Innereien Ihrer Floppy herumstöbern können. Nun ist der Speicherplatz bis auf 5 Byte ausgeschöpft, und die 4-KByte-Grenze soll auf keinen Fall überschritten werden. Wir haben daher andere Funktionen herausgenommen, und zwar für die Disassembler-Erweiterung den Diskmonitor und für die Diskmonitor-Erweiterung den Trace-Modus. Beide Erweiterungen sind also nicht gleichzeitig einsetzbar; überhaupt ist es sinnvoll, eigene Versionen für spezielle Anwendungen zusammenzustellen, eine »normale«, eine Spezial-Disk-Version und eine für verschärftes Disassemblieren.

Beginnen wir mit dem letzten: Wie Sie wissen, erscheinen beim Disassemblieren immer drei Sternchen, wenn SMON auf ein Byte trifft, das keinen gültigen 6510-Opcode darstellt. Nun wissen Sie aber vielleicht auch, daß es über den offiziel-

len Befehlssatz hinaus noch einige Befehle gibt, die der Hersteller des Prozessors zwar nicht dokumentiert hat, die aber nichtsdestotrotz funktionieren und in einigen Programmen auch ausgenutzt werden. Es wäre natürlich schön, wenn SMON auch diese »illegalen« Opcodes anzeigen könnte. Unsere Erweiterung macht's möglich.

Wir haben Mnemonics für eine Reihe dieser Befehle eingesetzt und lassen diese von SMON mit einem vorangestellten »*« ausgeben. Übrig bleiben noch zehn Befehle, deren Wirkung aber so komplex ist, daß sie sich beim besten Willen nicht mit einem Mnemonic abkürzen lassen. Sie fallen auch aus der Logik der Prozessorstruktur heraus. Im einzelnen handelt es sich um die Opcodes 0B, 2B, 4B, 6B, 8B, 9C, 9E, AB, CB und EB. Bei diesen Befehlen haben wir keine gemeinsame Struktur entdecken können. Die neuen Mnemonics haben folgende Bedeutung:

LAX	Load Akku and X entspricht LDA und LDX.
DCP	Decrement and ComPare entspricht DEC und CMP.
ISC	Increment and SubtraCt entspricht INC und SBC.
RLA	Rotate Left AND Akku entspricht ROL und AND.
RRA	Rotate Right and Add with carry entspricht ROR und ADC.
SLO	Shift Left OR Akku entspricht ASL und ORA.
SRE	Shift Right and EOR Akku entspricht LSR und EOR.
SAX	Store Akku AND X führt eine UND-Verknüpfung zwischen Akku und X-Register durch und speichert das Ergebnis in der angegebenen Adresse ab.
CRA	CRAsh führt zum »Absturz« des Prozessors.
NOP	NO Operation entspricht dem bekannten NOP, jedoch kann dieser Befehl auch 2 oder 3 Byte lang sein. Dies wird durch die angegebene Adresse deutlich, die in diesem Fall natürlich keinerlei Bedeutung hat.

Über den Sinn dieser Befehle läßt sich sicher streiten; allerdings kommen sie bisweilen in Programmen vor, meist um das Lesen dieser Programme unmöglich zu machen, also als Programmschutz. Von der Verwendung dieser Befehle in eigenen Programmen raten wir auf jeden Fall ab. Erstens wird kein Hersteller garantieren, daß die »illegalen« tatsächlich mit jedem 6510-Prozessor funktionieren, zweitens gibt es keine Funktion, die nicht auch mit den »normalen« Befehlen ebensogut erreicht werden könnte. Und als Programmschutz taugen die »illegalen« spätestens mit der Veröffentlichung dieses Artikels ja auch nichts mehr. Aus diesem Grund haben wir bewußt auf eine Erweiterung des Assemblers in dieser Richtung verzichtet. Sie können also keine normalen Opcodes durch Überschreiben in »illegale« ändern, wohl aber umgekehrt. Es bleibt lediglich die Eingabe als Einzelbyte, was aber hoffentlich zu umständlich ist.

Komfortabler Disketten-Monitor für SMON

Jetzt folgt unser zweiter Leckerbissen in Form eines kleinen aber ungemein wertvollen Zusatzprogrammes für den SMON. Es handelt sich dabei um eine Erweiterung des Disketten-Monitors, mit dem jeder auf einen Schlag die Arbeit von Stunden zunichte machen kann. Geben Sie das Programm wie beschrieben ein, starten Sie SMON wie gewohnt und springen mit »Z« in den Disketten-Monitor. Von hier aus erreichen Sie mit »F« (wie Floppy) die neuen Befehle. Wir haben absichtlich diesen umständlichen Weg gewählt, denn Fehler in diesem Modus wirken noch dramatischer als sonst. Mit diesem Werkzeug haben Sie unmittelbaren Zugriff auf die Eingeweide der Floppy. Jetzt können Sie die folgenden Befehle mit einer Übungsdiskette (!!!) in aller Ruhe durcharbeiten.

M Memory-Dump des Disketten-Monitors
Beispiel: M (ohne weitere Eingabe) listet den Bereich des Floppy-RAM von $0000-$00FF. (Es erscheint zunächst die erste Zeile, weitere Ausgabe mit der SPACE-Taste.)

In diesem Bereich befinden sich unter anderem die Jobspeicher ($00-$04) für die fünf Puffer 0 bis 4 sowie die wichtigsten Variablen des DOS.

M 07 Memory-Dump ab $0700
Die BAM der Diskette wird nach dem Initialisieren in Puffer 4 ($0700 im Floppy-RAM) eingelesen. Schauen Sie sich also mit »M 07« die aktuelle BAM an. Sie könnten jetzt durch einfaches Überschreiben den Inhalt der BAM ändern. (Der Doppelpunkt vor der Zeile wirkt als »hidden command«). Dann schauen Sie sich Ihre Änderung mit »M 07« wieder an. Sie sehen, daß inzwischen der Inhalt des Floppy-RAM geändert wurde. Wenn Sie nun den Jobcode »90« (= Schreibbefehl an den Floppy-Controller) in Speicherstelle $04 bringen, würde die geänderte (falsche!) BAM auf Diskette zurückgeschrieben werden!! Es gibt also genug Möglichkeiten, wie oben angedeutet, die Disketten zu »versauen«.

Für das Ausprobieren noch einige wichtige Speicherstellen und Jobcodes:

$80	Lesen
$90	Schreiben
$C0	»Anschlagen« des Kopfes
$D0	Maschinenprogramme im Puffer ausführen
$E0	Programm im Puffer ausführen mit Hochfahren des Laufwerks

Speicherstellen im Floppy-RAM:

$06/$07	ist Spur- und Sektornummer für den Befehl in Puffer 0
$08/$09	für Puffer 1
$0A/$0B	für Puffer 2
$0C/$0D	für Puffer 3
$0E/$0F	für Puffer 4

Jedem Puffer sind zwei Speicherstellen zugeordnet, eine für den Jobcode ($0000 bis $0004) und eine für Spur und Sektor. Wenn Sie also in Puffer 0 (in $0300 gelegen) einen bestimmten Block einlesen wollen, geben Sie folgende Befehle ein:

»M« liest die Zeropage der Floppy ein — so sehen dann zum Beispiel die ersten Zeilen aus:

```
:0000 01 01 01 FF 03 04 01 34
:0008 23 02 04 50 01 03 0A 11
```

Gehen Sie mit dem Cursor in die erste Zeile und schreiben Sie »80« in die erste Speicherstelle (anstelle der ersten 01). In Speicherstelle $06/$07 (die letzten beiden in der ersten Reihe) die Spur- und die Sektornummer, die gelesen werden soll, zum Beispiel 12 01. Sie sehen dann

```
:0000 80 01 01 FF 03 04 12 01
:0008 unverändert
```

Drücken Sie die RETURN-Taste, Mit »M 03« kann jetzt der eingelesene Block (hier der erste Directory-Block) angesehen werden. Änderungen können durch einfaches Überschreiben vorgenommen werden. Dauerhaft wird Ihre Änderung erst durch Zurückschreiben (nach Spur $12 und Sektor $01) mit dem Jobcode »90« in der ersten Speicherstelle. Nach Änderung der beiden für Puffer 0 zuständigen Adressen ($06/$07) auch an jede beliebige andere Stelle. Das ist wörtlich zu nehmen, denn wir befinden uns hier »unterhalb«

Befehlsübersicht zum SMON

Alle Eingaben erfolgen in der hexadezimalen Schreibweise. In Klammern angegebene Adreßeingaben können entfallen. SMON benutzt dann sinnvolle, vorgegebene Werte.

Bei allen Ausgabe-Befehlen ist gleichzeitig die Ausgabe auf einem Drucker möglich. Dazu werden die Befehle geSHIFTet eingegeben.

A **4000** (Assembler)
symbolischer Assembler (Verarbeitung von Label möglich) Startadresse $4000

B **4000 4200** (Basic-Data)
erzeugt Basic-DATA-Zeilen aus Maschinenprogramm im Bereich von $4000 bis $41FF

C **4010 4200 4013 4000 4200** (Convert)
in ein Programm, das von $4000 bis $4200 im Speicher steht, soll bei 4010 ein 3-Byte-Befehl eingefügt werden. Dazu wird das Programm ab $4010 bis 4200 auf die neue Adresse $4013 verschoben. Alle absoluten Adressen, die innerhalb des Programmbereichs ($4000 bis $4200) stehen, werden umgerechnet, so daß die Sprungziele stimmen.

D **4000 (4100)** (Disassembler)
disassembliert den Bereich von $4000 (bis $4100) mit Ausgabe der Hex-Werte. Änderungen sind durch Überschreiben der Befehle möglich.

F (Find)
findet Zeichenketten (F), absolute Adressen (FA), relative Sprünge (FR), Tabellen (FT), Zeropageadressen (FZ) und Immediate-Befehle (FI)

G **(4000)** (Go)
startet ein Maschinenprogramm, das bei $4000 im Speicher beginnt

I **01** (I/O-Gerät)
stellt die Gerätenummer für Floppy (08 oder 09) oder Datasette (01) ein

K **A000 (A500)** (Kontrolle)
zum schnellen Durchsuchen des Bereichs von $A000 (bis $A500) nach ASCII-Zeichen (32 Byte pro Zeile). Änderungen sind durch Überschreiben der ASCII-Zeichen möglich.

L **(4000)** (Load)
lädt ein Maschinenprogramm an die richtige oder eine angegebene Adresse ($4000)

M **4000 (4400)** (Memory-Dump)
gibt den Inhalt des Speichers von $4000 (bis $43FF) in Hex-Byte und ASCII-Code aus. Änderungen sind durch Überschreiben der Hex-Zahlen möglich.

O **4000 4500 AA** (Occupy)
füllt den Speicherbereich von $4000 bis $4500 mit vorgegebenem Byte ($AA) aus

P **05** (Printer)
setzt Geräteadresse 5 für Drucker

R (Register)
zeigt die Registerinhalte und Flags an. Änderungen sind durch Überschreiben möglich.

S **"Test" 4000 5000** (Save)
speichert ein Programm von $4000 bis $4FFF unter dem Namen »Test« ab

TW **(4000)** (Trace Walk)
führt auf Tastendruck den jeweils nächsten Maschinenbefehl aus und zeigt die Registerinhalte an. Subroutinen können in Echtzeit durchlaufen werden (»J«). Wird keine Startadresse eingegeben, beginnt »TW« bei der letzten mit »R« angezeigten Adresse.

TB **4010 05** (Trace Break)
setzt einen Haltepunkt für den Schnellschrittmodus bei $4010. Der Schnellschrittmodus wird unterbrochen, nachdem $4010 zum fünften Mal erreicht worden ist.

TQ **4000** (Trace quick)
Schnellschrittmodus, springt beim Erreichen eines Haltepunktes in den Einzelschrittmodus

TS **4000 4020** (Trace stop)
arbeitet ein Programm ab $4000 in Echtzeit ab und springt beim Erreichen von $4020 in die Registeranzeige. Von dort aus kann (nach eventueller Änderung der Register) mit »G« oder »TW« fortgefahren werden. »TS« arbeitet nur im RAM-Speicher.

V **6000 6200 4000 4100 4200** (Verschieben)
ändert in einem Programm von $4100 bis $41FF alle absoluten Adressen, die sich auf den Bereich von $6000 bis $6200 beziehen, auf einen neuen Bereich, der bei $4000 beginnt.

W **4000 4300 5000** (Write)
verschiebt den Speicherinhalt von $4000 bis $42FF nach $5000 ohne Umrechnung der Adressen (zum Beispiel Tabellen)

X (Exit)
springt aus dem Monitor-Programm ins Basic zurück

**49152**
Dezimalzahl umrechnen

$ **002B**
4stellige Hex-Zahl umrechnen

% **01101010**
8stellige Binärzahl umrechnen

? **0344 + 5234**
Addition oder Subtraktion zweier 4stelliger Hex-Zahlen

= **4000 5000** (Vergleich)
vergleicht den Speicherinhalt ab $4000 mit dem ab $5000

Z (Diskmonitor)
ruft den Diskmonitor auf. Dieser verfügt über folgende Befehle:

R **(12 01)** (Read)
liest Track $12, Sektor $01 von der Diskette in einen Puffer im Speicher. Fehlt die Angabe von Track und Sektor, wird der logisch (!) nächste Sektor gelesen.

W **(12 01)** (Write)
schreibt den Puffer im Speicher nach Track $12, Sektor $01 auf die Diskette. Ohne Angabe von Track und Sektor werden die letzten Eingaben von »R« benutzt.

M (Memory-Dump)
zeigt den Pufferinhalt als Hexdump (wie normales »M«). Weitere Ausgabe mit CBM-Taste, Abbruch mit STOP. Werte können durch Überschreiben geändert werden.

X (Exit)
springt in SMON zurück

F (weitere Disketten-Befehle initialisieren)
sind die Befehle initialisiert, gilt:

- **M** (07)
  Memory-Dump (Floppy-RAM/ROM)
- **V** 6000 0400
  Verschieben eines 256-Byte-Blocks von $6000 in den Laufwerkspuffer 1 beziehungsweise in das Floppy-RAM
- **@**
  normale Disketten-Befehle senden
- **X**
  zurück zum normalen Disketten-Monitor

der Controllerebene, die unter anderem für die Prüfung auf Einhaltung der zulässigen Spur und Sektorgrenzen verantwortlich ist. Es erfolgt also keine Fehlermeldung, wenn Sie versuchen sollten, mit Ihrer Floppy bis in die des Nachbarn zu schreiben (zum Beispiel mit der Spur 152).

Entsprechende Lese- und Schreibübungen können mit den anderen Puffern durchgeführt werden. Denken Sie daran, erst ist die Spur- beziehungsweise Sektornummer für den entsprechenden Puffer (in der zweiten Zeile!) einzugeben, bevor Sie in Zeile 1 den Jobcode mit einem »RETURN« übergeben, denn mit Druck auf die RETURN-Taste wird Ihr Befehl ausgeführt. Und noch eins: Quälen Sie bitte dabei Ihren Schreibkopf nicht mehr als unbedingt erforderlich, sonst könnte er sich mechanisch verklemmen und nur noch mit einem Eingriff in die Floppymechanik wieder »befreit« werden.

Falls Sie die Ausgaben 1/85 (Seite 151) und 3/85 (Seite 103 bis 135) der 64'er besitzen, können Sie sich dort über andere Speicherstellen der Floppy und die weitere Anwendung der Jobcodes informieren.

Der Befehl @ ohne weitere Angaben fragt den Fehlerkanal ab, ansonsten dient er zur Befehlsübermittlung an die Floppy.

Beispiel:	@	Fehlerkanal
	@I	Initialisierungsbefehl oder
	@S:name	Befehl zum Scratchen
		und so weiter.

Bedingt durch die verschiedenen Versionen, springt dieser Befehl manchmal in den »normalen« Disketten-Monitor zurück, erkennbar an dem »*« am Zeilenanfang. Sie müssen dann wieder ein »F« eingeben.

Mit X gelangt man wieder in den Disketten-Monitor.

Zum Abschluß ein sehr hilfreicher Befehl namens »V«, der es erlaubt, Speicherbereiche aus dem Computer in den Laufwerkspuffer zu verschieben. Folgende einfache Syntax gilt dabei: **V von nach**

Um zum Beispiel ein Maschinenprogramm von $6000 in den Puffer 1 zu bekommen, geben Sie folgendes ein:
V 6000 0400

Dabei wird immer eine ganze Seite, also 256 Byte, übertragen. Was das Programm dort soll, fragen Sie? Führen Sie es doch einfach aus (Jobcode $D0 in Speicherstelle $01 schreiben); oder schreiben Sie es mit dem Jobcode »90« in einen beliebigen Sektor der Diskette.

Wenn Sie dann Ihre Floppy so richtig durcheinander gebracht haben und nichts läuft mehr, brauchen Sie nicht zu verzweifeln. Außer einem eventuell festhängenden Lesekopf passiert der Floppy nichts, nur Ihren Disketten.

Hinweise zum Abtippen

Tippen Sie die beiden Erweiterungsprogramme (Listing 2 und 3 beziehungsweise bei der M&T-Version Listing 4 und 5) mit dem MSE-Programm ab und speichern Sie die fertigen Programme. Die Programme für die M&T-Version haben sinnigerweise ein M&T im Namen.

Laden und starten Sie dann Ihren SMON $C000. Geben Sie ein: L"NDISASS"

Damit werden die neuen Befehle automatisch über den bisherigen Disketten-Monitor geladen. Sie müssen nun aber noch aktiviert werden. Geben Sie dazu G CF0D ein.

SMON meldet sich sofort mit seiner Registeranzeige wieder. Sie sollten nun diese Version unbedingt abspeichern, zum Beispiel mit S"SMON NDISASS" C000 CF3D

Wenn Sie nun das Programm »ILLEGAL-CODE« (Listing 6) laden und mit D 4000 disassemblieren, sehen Sie die »illegalen« Opcodes schön geordnet nacheinander.

Um die neuen Befehle des Disketten-Monitors in SMON einzubinden, gehen Sie ganz ähnlich vor. Nach dem Abtippen und Speichern des Programms »FLOPPYMON« muß natürlich SMON C000 geladen und gestartet werden. Anschließend geben Sie ein: L"FLOPPYMON" und aktivieren es mit G CDD8 (64'er-Version) beziehungsweise G CDB6 (M&T-Version).

Zum Speichern geben Sie S"SMON-FLOPPY"COOO CFFF ein. Das gilt für beide Versionen.

(Dietrich Weineck/ah)

SMON-Speicherstellen

Folgende Zeropage-Adressen werden benutzt:

FLAG	$AA	Universalflag
ADRCODE	$AB	Adressierungscode für Assembler/Disassembler
COMMAND	$AC	SMON-Befehlscode
BEFCODE	$AD	Befehlscode Ass./Disass.
LOPER	$AE	Low-Operand für Ass./Disass.
HOPER	$AF	High-Operand für Ass./Disass.
BEFLEN	$B6	Befehlslänge Ass./Disass.
PCL	$FB	SMON-Programmcounter Low-Byte
PCH	$FC	SMON-Programmcounter High-Byte

Außerhalb der Zeropage benutzt SMON die Bereiche:

PCHSAVE	$02A8	
PCLSAVE	$02A9	
SRSAVE	$02AA	
AKSAVE	$02AB	dienen der Zwischenspeicherung
XRSAVE	$02AC	der angegebenen Register
YRSAVE	$02AD	
SPSAVE	$02AE	
PRINTER	$02AF	Printernummer
IO.NR	$02B0	Devicenummer
MEM	$02B1	Buffer bis $02B8
TRACEBUF	$02B8 bis $02BF	Buffer für Trace-Modus

Dann folgen die von Diskmonitor benötigten Adressen:

SAVEX	$02C1	Zwischenspeicherung der X- und Y-Register
TMPTRCK	$02C2	
TMPSECTO	$02C3	Zwischenspeicher für Track und Sektor
DCMDST	$02D0	Diskkommandostring
TRACK	$02D8	
SECTO	$02DB	Track und Sektornummer
BUFFER	$033C bis $03FC	Buffer für Label, nur für Assembler

Einsprungadressen von SMON-Routinen

Die Angaben in Klammern beziehen sich auf die M&T-Version

; (TICK)	$CADB	($CACF)	
# (BEFDEC)	$C92E	($C92F)	
$ (BEFHEX)	$C908	($C909)	
% (BEFBIN)	$C91C	($C910)	
, (KOMMA)	$C6FC	($C6B1)	
: (COLON)	$C41D	($C40B)	
; (SEMIS)	$C3B6	($C3A0)	
= (COMP)	$CAF5	($CAE9)	– V Kommando
? (ADDSUB)	$C89A	($C89B)	
A (ASSEMBLER)	$C6D1	($C6BC)	
B (BASICDATA)	$C96C	($C96D)	
C (CONVERT)	$CA3D	($CA32)	

D (DISASS.)	$C55D	($C542)	P (SETPRINTER)	$C83D	($C829)
F (FIND)	$CB11	($CB0C)	R (REGISTER)	$C386	($C370)
G (GO)	$C3E3	($C3CD)	S (LOADSAVE)	$C84E	($C83A)
I (IO.SET)	$C844	($C830)	T (TRACE)	$CBF1	($CBEC)
K (KONTROLLE)	$CAB7	($CAAC)	V (VERSCHIEB)	$CA43	($CA38) – U Kommando
L (LOADSAVE)	$C84E	($C83A)	W (WRITE)	$C9D3	($C9D4)
M (MEMDUMP)	$C3F9	($C3E3)	X (EXIT)	$C36E	($C369)
O (OCUPPY)	$C9C1	($C9C2)	Z (DMON)	$CE09	($CDFE)

```
programm : smon $c000          c000 cffa
-------------------------------------------

c000 : a9 14 8d 16 03 a9 c2 8d   7d
c008 : 17 03 00 27 23 24 25 2c   c6
c010 : 3a 3b 3d 3f 41 42 43 44   db
c018 : 46 47 49 4b 4c 4d 4f 50   cb
c020 : 52 53 54 56 57 58 5a 00   9d
c028 : 00 00 00 da ca 2d c9 07   cf
c030 : c9 1b c9 fb c6 1c c4 b5   44
c038 : c3 f4 ca 99 c8 d0 c6 6b   60
c040 : c9 3c ca 5c c5 10 cb e2   37
c048 : c3 43 c8 b6 ca 4d c8 f8   e2
c050 : c3 c0 c9 3c c8 85 c3 4d   d0
c058 : c8 f0 cb 42 ca d2 c9 6d   19
c060 : c3 08 ce 00 00 00 00 00   db
c068 : 00 00 00 ff ff 01 00 41   f3
c070 : 5a 49 52 54 80 20 40 10   b8
c078 : 00 02 01 01 02 00 91 91   63
c080 : 0d 53 d9 31 37 32 0d 00   0d
c088 : 7d 4c 7d c9 0d 0d 20 20   be
c090 : 50 43 20 20 53 52 20 41   59
c098 : 43 20 58 52 20 59 52 20   a2
c0a0 : 53 50 20 20 4e 56 2d 42   f8
c0a8 : 44 49 5a 43 00 02 04 01   b2
c0b0 : 2c 00 2c 59 29 58 9d 1f   1d
c0b8 : ff 1c 1c 1f 1f 1f 1c df   cb
c0c0 : 1c 1f df ff ff 03 1f 80   f9
c0c8 : 09 20 0c 04 10 01 11 14   da
c0d0 : 96 1c 19 94 be 6c 03 13   cf
c0d8 : 01 02 02 03 03 02 02 02   08
c0e0 : 02 02 02 03 03 02 03 03   17
c0e8 : 03 02 00 40 40 80 80 20   3f
c0f0 : 10 25 26 21 22 81 82 21   bb
c0f8 : 82 84 08 08 e7 e7 e7 e7   ed
c100 : e3 e3 e3 e3 e3 e3 e3 e3   ff
c108 : e3 e3 e7 a7 e7 e7 f3 f3   41
c110 : f7 df 26 46 06 66 41 81   e5
c118 : e1 01 a0 a2 a1 c1 21 61   66
c120 : 84 86 e6 c6 e0 c0 24 4c   b7
c128 : 20 90 b0 f0 30 d0 10 50   45
c130 : 70 78 00 18 d8 58 b8 ca   a8
c138 : 88 e8 c8 ea 48 08 68 28   7a
c140 : 40 60 aa a8 ba 8a 9a 98   0c
c148 : 38 f8 89 9c 9e b2 2a 4a   af
c150 : 0a 6a 4f 23 93 b3 f3 33   d5
c158 : d3 13 53 73 52 4c 41 52   29
c160 : 45 53 53 4f 4c 4c 4c 43   ec
c168 : 41 41 53 53 49 44 43 43   d3
c170 : 42 4a 4a 42 42 42 42 42   76
c178 : 42 42 42 53 42 43 43 43   a8
c180 : 43 44 44 49 49 4e 50 50   09
c188 : 50 50 52 52 54 54 54 54   c1
c190 : 54 54 53 53 4f 53 53 4f   c9
c198 : 4f 54 42 52 44 44 44 4d   fe
c1a0 : 4e 44 54 54 4e 45 50 50   a1
c1a8 : 49 4d 53 43 43 45 4d 4e   05
c1b0 : 50 56 56 45 52 4c 4c 4c   bb
c1b8 : 4c 45 45 4e 4e 4f 48 48   d3
c1c0 : 4c 4c 54 54 41 41 53 58   ee
c1c8 : 58 59 45 45 4c 52 4c 52   f4
c1d0 : 52 41 43 41 59 58 41 50   ba
c1d8 : 44 43 59 58 43 43 58 59   82
c1e0 : 54 50 52 43 53 51 49 45   c9
c1e8 : 4c 43 53 49 4b 43 44 49   46
c1f0 : 56 58 59 58 59 50 41 50   91
c1f8 : 41 50 49 53 58 59 58 41   52
c200 : 53 41 43 44 08 84 81 22   3c
c208 : 21 26 20 80 03 20 1c 14   1e
c210 : 14 10 04 0c d8 a9 08 8d   c5
c218 : b0 02 a9 04 8d af 02 a9   66
c220 : 06 8d 20 d0 8d 21 d0 a9   87
c228 : 03 8d 86 02 a2 05 68 9d   03
c230 : a8 02 ca 10 f9 ad a9 02   46
c238 : d0 03 ce a8 02 ce a9 02   94
c240 : ba 8e ae 02 a9 52 4c ff   8c
c248 : c2 20 c2 c2 f0 0b 20 7e   08
c250 : c2 8d a9 02 a5 fc 8d a8   4d
c258 : 02 60 a2 a4 20 80 c2 20   19
c260 : 80 c2 d0 1c 20 7e c2 a9   4d
c268 : fe 85 fd a9 ff 85 fe 20   46
c270 : c2 c2 d0 0c 8d 77 02 e6   b3
c278 : c6 60 20 7e c2 2c a2 fb   56
c280 : 20 8d c2 95 01 20 9a c2   cb
c288 : 95 00 e8 e8 60 20 ca c2   2c
c290 : c9 20 f0 f9 c9 2c f0 f5   92
c298 : d0 03 20 ca c2 20 af c2   bd
c2a0 : 0a 0a 0a 0a 85 b4 20 ca   87
c2a8 : c2 20 af c2 05 b4 60 c9   ca
c2b0 : 3a 90 02 69 08 29 0f 60   a7
c2b8 : 20 ca c2 c9 20 f0 f9 c6   26
c2c0 : d3 60 20 cf ff c6 d3 c9   de
c2c8 : 0d 60 20 cf ff c9 0d d0   2b
c2d0 : f8 a9 3f 20 d2 ff ae ae   b6
c2d8 : 02 9a a2 00 86 c6 20 51   92
c2e0 : c3 a1 d1 c9 27 f0 11 c9   f3
c2e8 : 3a f0 0d c9 3b f0 09 c9   0a
c2f0 : 2c f0 05 a9 2e 20 d2 ff   3a
c2f8 : 20 ca c2 c9 2e f0 f9 85   c4
c300 : ac 29 7f a2 20 dd 0a c0   10
c308 : f0 05 ca d0 f8 f0 c2 20   aa
c310 : 15 c3 4c d6 c2 8a 0a aa   f3
c318 : e8 bd 29 c0 48 ca bd 29   65
c320 : c0 48 60 a5 fc 20 2a c3   d2
c328 : a5 fb 48 4a 4a 4a 4a 20   87
c330 : 35 c3 68 29 0f c9 0a 90   0f
c338 : 02 69 06 69 30 4c d2 ff   4e
c340 : a9 0d 20 d2 ff 8a 4c d2   fd
c348 : ff 20 4c c3 a9 20 4c d2   55
c350 : ff a9 0d 4c d2 ff 85 bb   ab
c358 : 84 bc a0 00 b1 bb f0 06   2b
c360 : 20 d2 ff c8 d0 f6 60 e6   16
c368 : fb d0 02 e6 fc [illegible] [illegible] 0e   be
c370 : 8d 86 02 8d 20 d0 a9 06   ae
c378 : 8d 21 d0 a9 37 85 01 ae   00
c380 : ae 02 9a 4c 74 a4 a0 c0   d0
c388 : a9 8c 20 56 c3 a2 3b 20   c9
c390 : 40 c3 ad a8 02 85 fc ad   ce
c398 : a9 02 85 fb 20 23 c3 20   8e
c3a0 : 4c c3 a2 fb bd af 01 20   93
c3a8 : 2a c3 20 4c c3 e8 d0 f4   f6
c3b0 : ad aa 02 4c d0 c3 20 4e   05
c3b8 : c2 a2 fb 20 ca c2 20 9a   47
c3c0 : c2 9d af 01 e8 d0 f4 20   86
c3c8 : 4c c3 bd aa 02 4c d0 c3   08
c3d0 : 85 aa a9 20 a0 09 20 d2   91
c3d8 : ff 06 aa a9 30 69 00 88   19
c3e0 : d0 f4 60 20 49 c2 ae ae   09
c3e8 : 02 9a a2 fa bd ae 01 48   25
c3f0 : e8 d0 f9 68 a8 68 aa 68   15
c3f8 : 40 20 64 c2 a2 3a 20 40   b7
c400 : c3 20 23 c3 a0 20 a2 00   aa
c408 : 20 4c c3 a1 fb 20 2a c3   64
c410 : a1 fb 20 39 c4 d0 f1 20   b9
c418 : 5d c4 90 e0 60 20 7e c2   9e
c420 : a0 20 a2 00 20 ca c2 20   1d
c428 : 9a c2 81 fb c1 fb f0 03   c9
c430 : 4c d1 c2 20 39 c4 d0 ec   f0
c438 : 60 c9 20 90 0c c9 60 90   49
c440 : 0a c9 c0 90 04 c9 db 90   90
c448 : 04 a9 2e 29 3f 29 7f 91   30
c450 : d1 ad 86 02 91 f3 20 67   e2
c458 : c3 c8 c0 28 60 20 6f c4   03
c460 : 4c 66 c4 20 67 c3 a5 fb   38
c468 : c5 fd a5 fc e5 fe 60 20   4d
c470 : 94 c4 20 86 c4 f0 0e 20   8b
c478 : 86 c4 f0 fb c9 20 d0 05   07
c480 : 8d 77 02 e6 c6 60 20 e4   e0
c488 : ff 48 20 e1 ff f0 02 68   50
c490 : 60 4c d6 c2 a0 28 24 ac   59
c498 : 10 f6 84 c8 84 d0 a9 ff   d3
c4a0 : 20 c3 ff a9 ff 85 b8 85   f1
c4a8 : b9 ad af 02 85 ba 20 c0   94
c4b0 : ff a2 00 86 d3 ca 20 c9   79
c4b8 : ff 20 cf ff 20 d2 ff c9   e7
c4c0 : 0d d0 f6 20 cc ff a9 91   8d
c4c8 : 4c d2 ff a0 00 b1 fb 24   57
c4d0 : aa 30 02 50 0c a2 1f dd   2b
c4d8 : 3c c1 f0 2f ca e0 15 d0   c0
c4e0 : f6 a2 04 dd 49 c1 f0 21   8d
c4e8 : dd 4d c1 f0 1e ca d0 f3   5e
c4f0 : a2 38 dd 11 c1 f0 14 ca   d1
c4f8 : e0 16 d0 f6 b1 fb 3d fb   de
c500 : c0 5d 11 c1 f0 05 ca d0   ef
c508 : f3 a2 00 86 ad 8a f0 0f   2e
c510 : a2 11 b1 fb 3d b5 c0 5d   66
c518 : c6 c0 f0 03 ca d0 f3 bd   59
c520 : ea c0 85 ab bd d8 c0 85   f2
c528 : b6 a6 ad 60 a0 01 b1 fb   7a
c530 : aa c8 b1 fb a0 10 c4 ab   1f
c538 : d0 07 20 4a c5 a0 03 d0   ec
c540 : 02 a4 b6 8e ae 00 8d af   94
c548 : 00 60 a0 01 b1 fb 10 01   fe
c550 : 88 38 65 fb aa e8 f0 01   85
c558 : 88 98 65 fc 60 a2 00 86   4d
c560 : aa 20 64 c2 20 8c c5 a5   54
c568 : ad c9 16 f0 09 c9 30 f0   1f
c570 : 05 c9 21 d0 11 ea 20 94   ce
c578 : c4 20 51 c3 a2 23 a9 2d   5d
c580 : 20 d2 ff ca d0 fa 20 5d   83
c588 : c4 90 d9 60 a2 2c 20 40   a3
c590 : c3 20 23 c3 20 4c c3 20   58
c598 : 75 c6 20 cb c4 20 4c c3   f8
c5a0 : b1 fb 20 2a c3 20 4c c3   92
c5a8 : c8 c4 b6 d0 f3 a9 03 38   a3
c5b0 : e5 b6 aa f0 09 20 49 c3   f7
c5b8 : 20 4c c3 ca d0 f7 a9 20   fc
c5c0 : 20 d2 ff a0 00 a6 ad d0   eb
c5c8 : 11 a2 03 a9 2a 20 d2 ff   0f
c5d0 : ca d0 f8 24 aa 30 85 4c   a0
c5d8 : 6a c6 24 aa 50 29 a9 08   09
c5e0 : 24 ab f0 23 b1 fb 29 fc   14
c5e8 : 85 ad c8 b1 fb 0a a8 b9   d2
c5f0 : 3c 03 8d ae 00 c8 b9 3c   8d
c5f8 : 03 8d af 00 20 be c6 a4   0a
c600 : b6 20 93 c6 20 cb c4 bd   73
c608 : 5b c1 20 d2 ff bd 93 c1   66
c610 : 20 d2 ff bd cb c1 20 d2   42
c618 : ff a9 20 24 ab f0 03 20   07
c620 : 49 c3 a2 20 a9 04 24 ab   9a
c628 : f0 02 a2 28 8a 20 d2 ff   bc
c630 : 24 ab 50 05 a9 23 20 d2   b8
c638 : ff 20 2c c5 88 f0 16 a9   c7
c640 : 08 24 ab f0 07 a9 4d 20   96
c648 : d2 ff a0 01 b9 ad 00 20   ab
c650 : 2a c3 88 d0 f7 a0 03 b9   9c
c658 : ac c0 24 ab f0 09 b9 af   80
c660 : c0 be b2 c0 20 42 c3 88   78
c668 : d0 ed a5 b6 20 67 c3 38   2c
c670 : e9 01 d0 f8 60 a4 d3 a9   fb
c678 : 20 91 d1 c8 c0 28 90 f9   72
c680 : 60 e4 ab d0 04 05 ad 85   81
c688 : ad 60 b9 ad 00 91 fb d1   a9
c690 : fb d0 04 88 10 f4 60 68   00
c698 : 68 60 d0 1c 8a 05 ab 85   72
c6a0 : ab a9 04 85 b5 20 cf ff   6d
c6a8 : c9 20 f0 0d c9 24 f0 09   f3
c6b0 : c9 28 f0 05 c9 2c f0 01   2e
c6b8 : 60 c6 b5 d0 e8 60 e0 18   48
c6c0 : 30 0e ad ae 00 38 e9 02   a6
c6c8 : 38 e5 fb 8d ae 00 a0 40   91
c6d0 : 60 20 7e c2 85 fd a5 fc   11
c6d8 : 85 fe 20 51 c3 20 e4 c6   6d
c6e0 : 30 fb 10 f6 a9 00 85 d3   49
c6e8 : 20 4c c3 20 23 c3 20 4c   8d
c6f0 : c3 20 cf ff a9 01 85 d3   17
c6f8 : a2 80 d0 05 a2 80 8e b1   7b
c700 : 02 86 aa 20 7e c2 a9 25   e3
c708 : 85 c8 2c b1 02 10 08 a2   39
c710 : 0a 20 cf ff ca d0 fa a9   90
c718 : 00 8d b1 02 20 a1 c6 c9   49
c720 : 46 d0 16 46 aa 68 68 a2   f2
c728 : 02 b5 fa 48 b5 fc 95 fa   5c
c730 : 68 95 fc ca d0 f3 4c 64   a2
c738 : c5 c9 2e d0 11 20 9a c2   89
c740 : a0 00 91 fb d1 fb d0 04   0c
c748 : 20 67 c3 c8 88 60 a2 fd   38
c750 : c9 4d d0 19 20 9a c2 a0   3a
c758 : 00 c9 3f b0 ef 0a a8 a5   60
c760 : fb 99 3c 03 a5 fc c8 99   30
c768 : 3c 03 20 a1 c6 95 a9 e0   e4
c770 : fd d0 04 a9 07 85 b7 e8   59
c778 : d0 f0 a2 38 a5 a6 dd 5b   2e
```

```
c780 : c1 f0 05 ca d0 f6 ca 60  05
c788 : a5 a7 dd 93 c1 d0 f4 a5  ac
c790 : a8 dd cb c1 d0 ed bd 11  e7
c798 : c1 85 ad 20 a1 c6 a0 00  5e
c7a0 : e0 20 10 09 c9 20 d0 08  a7
c7a8 : bd 4d c1 85 ad 4c 31 c8  c0
c7b0 : a0 08 c9 4d f0 20 a0 40  83
c7b8 : c9 23 f0 1a 20 9d c2 8d  a7
c7c0 : ae 00 8d af 00 20 a1 c6  dd
c7c8 : a0 20 c9 30 90 1b c9 47  88
c7d0 : b0 17 a0 80 c6 d3 20 a1  13
c7d8 : c6 20 9d c2 8d ae 00 20  fd
c7e0 : a1 c6 c0 08 f0 03 20 be  3b
c7e8 : c6 84 ab a2 01 c9 58 20  2f
c7f0 : 9a c6 a2 04 c9 29 20 9a  b2
c7f8 : c6 a2 02 c9 59 20 9a c6  58
c800 : a5 ad 29 0d f0 0a a2 40  d2
c808 : a9 08 20 81 c6 a9 18 2c  60
c810 : a9 1c ad a2 82 20 81 c6 a0  2b
c818 : 08 a5 ad c9 20 f0 09 be  c3
c820 : 03 c2 b9 0b c2 20 81 c6  15
c828 : 88 d0 f4 a5 ab 10 01 c8  db
c830 : c8 20 8a c6 c6 b7 a5 b7  b4
c838 : 85 d3 4c 97 c5 20 8d c2  c6
c840 : 8d af 02 60 20 8d c2 8d  c6
c848 : b0 02 60 4c d1 c2 a0 02  55
c850 : 84 bc 88 84 b9 84 bb 88  a5
c858 : 84 b7 20 ca c2 c9 22 d0  be
c860 : ea 20 ca c2 91 bb c8 e6  4d
c868 : b7 c9 22 d0 f4 c6 b7 ad  66
c870 : b0 02 85 ba a5 ac c9 53  67
c878 : f0 13 20 c2 c2 f0 09 a2  6f
c880 : c3 20 80 c2 a9 00 85 b9  f0
c888 : a9 00 6c 30 03 a2 c1 20  df
c890 : 80 c2 a2 ae 20 80 c2 6c  da
c898 : 32 03 20 7e c2 20 ca c2  02
c8a0 : 49 02 4a 4a 08 20 80 c2  cf
c8a8 : 20 51 c3 28 b0 0c a5 fd  65
c8b0 : 65 fb aa a5 fe 65 fc 38  f2
c8b8 : b0 09 a5 fb e5 fd aa a5  1a
c8c0 : fc e5 fe a8 8a 84 fc 85  4f
c8c8 : fb 84 62 85 63 08 a9 00  6c
c8d0 : 85 d3 20 75 c6 a5 fc d0  25
c8d8 : 0f 20 49 c3 a5 fb 20 2a  d1
c8e0 : c3 a5 fb 20 d0 c3 f0 03  6e
c8e8 : 20 23 c3 20 4c c3 a2 90  1d
c8f0 : a5 01 8d b1 02 a9 37 85  05
c8f8 : 01 28 20 49 bc 20 dd bd  fe
c900 : ae b1 02 86 01 4c 56 c3  2c
c908 : 20 8d c2 aa a4 d3 b1 d1  48
c910 : 49 20 f0 a3 8a a8 20 9a  bd
c918 : c2 38 b0 a9 20 b8 c2 a0  6c
c920 : 08 48 20 ca c2 c9 31 68  be
c928 : 2a 88 d0 f5 f0 eb 20 b8  e9
c930 : c2 a2 00 8a 86 fb 85 fc  ed
c938 : a8 20 cf ff c9 3a b0 84  1e
c940 : e9 2f b0 04 38 4c c4 c8  f8
c948 : 85 fd 06 fb 26 fc a5 fc  a8
c950 : 85 fe a5 fb 0a 26 fe 0a  1f
c958 : 26 fe 18 65 fb 08 18 65  db
c960 : fd aa a5 fe 65 fc 28 69  ad
c968 : 00 4c 34 c9 20 7a c2 a9  09
c970 : 37 85 01 a2 04 bd 87 c0  cc
c978 : 95 aa ca 10 f8 20 51 c3  74
c980 : a6 aa a5 ab 20 cd bd e6  8f
c988 : aa d0 02 e6 ab a9 44 20  51
c990 : d2 ff a9 c1 20 d2 ff a0  de
c998 : 00 b1 fb 84 62 85 63 20  20
c9a0 : d1 bd 20 63 c4 a2 03 b0  93
c9a8 : 0a a9 2c a6 d3 e0 49 90  f1,
c9b0 : e3 a2 09 86 c6 bd 7d c0  c9
c9b8 : 9d 76 02 ca d0 f7 4c 6e  45
c9c0 : c3 20 7a c2 20 8d c2 a2  49
c9c8 : 00 81 fb 48 20 63 c4 68  92
c9d0 : 90 f7 60 20 5a c2 a5 a6  18
c9d8 : d0 02 c6 a7 c6 a6 20 30  d2
c9e0 : ca 86 b5 a0 02 90 04 a2  69
c9e8 : 02 a0 00 18 a5 a6 65 ae  c0
c9f0 : 85 aa a5 a7 65 af 85 ab  6a
c9f8 : a1 a4 81 a8 41 a8 05 b5  3a
ca00 : 85 b5 a5 a4 c5 a6 a5 a5  d1
ca08 : e5 a7 b0 1d 18 b5 a4 79  45
ca10 : 6b c0 95 a4 b5 a5 79 6c  1c
ca18 : c0 95 a5 8a 18 69 04 aa  90
ca20 : c9 07 90 e8 e9 08 aa b0  99
ca28 : cf a5 b5 f0 0f 4c d1 c2  75
ca30 : 38 a2 fe b5 aa f5 a6 95  50
ca38 : b0 e8 d0 f7 60 20 62 ca  b5
ca40 : 4c d6 c9 4c 62 ca c5 a7  d6
ca48 : d0 02 e4 a6 b0 13 c5 a5  2d
ca50 : d0 02 e4 a4 90 0b 85 b4  d0
ca58 : 8a 18 65 ae aa a5 b4 65  93
ca60 : af 60 20 5a c2 20 7a c2  2f
ca68 : 20 30 ca 20 cb c4 c8 a9  b0
ca70 : 10 24 ab f0 26 a6 fb a5  6e
ca78 : fc 20 46 ca 86 aa b1 fb  ec
ca80 : 85 b5 20 4a c5 a0 01 20  d7
ca88 : 46 ca ca 8a 18 e5 aa 91  b6
ca90 : fb 45 b5 10 19 20 51 c3  fd
ca98 : 20 23 c3 24 ab 10 0f b1  9a
caa0 : fb aa c8 b1 fb 20 46 ca  c8
caa8 : 91 fb 8a 88 91 fb 20 6a  39
cab0 : c6 20 66 c4 90 b5 60 20  31
cab8 : 64 c2 a2 27 20 40 c3 20  5e
cac0 : 23 c3 a0 08 a2 00 20 4c  31
cac8 : c3 a1 fb 20 39 c4 d0 f9  50
cad0 : a2 00 20 5d c4 f0 03 4c  9f
cad8 : ba ca 60 20 7e c2 a0 03  9a
cae0 : 20 cf ff 88 d0 fa 20 ca  f4
cae8 : c2 c9 2e f0 02 91 fb c8  67
caf0 : c0 20 90 f2 60 20 7a c2  b9
caf8 : a2 00 a1 fb c1 fd d0 0b  e7
cb00 : 20 67 c3 e6 fd d0 f3 e6  a5
cb08 : fe d0 ef 20 4c c3 4c 23  c9
cb10 : c3 a9 ff a2 04 95 fa ca  6a
cb18 : d0 fb 20 ca c2 a2 05 dd  58
cb20 : 6e c0 f0 45 ca d0 f8 86  f7
cb28 : a9 20 b4 cb e8 20 cf ff  57
cb30 : c9 20 f0 f3 c9 2c d0 03  0b
cb38 : 20 7a c2 20 51 c3 a4 a9  63
cb40 : b1 fb 20 d6 cb d0 18 88  86
cb48 : 10 f6 20 23 c3 20 4c c3  36
cb50 : a4 d3 c0 24 90 09 20 94  8d
cb58 : c4 20 72 c4 20 51 c3 20  3d
cb60 : 63 c4 90 da a0 27 4c 96  46
cb68 : c4 bd 73 c0 85 a8 bd 78  85
cb70 : c0 85 a9 aa f0 06 20 b4  dc
cb78 : cb ca d0 fa 20 7a c2 20  5d
cb80 : cb c4 20 2c c5 a5 a8 24  af
cb88 : ab d0 09 a8 d0 21 a5 ad  fb
cb90 : d0 1d f0 0d a4 a9 b9 ad  a6
cb98 : 00 20 d6 cb d0 11 88 d0  31
cba0 : f5 84 aa 20 8c c5 20 6f  dc
cba8 : c4 20 66 c4 90 d1 60 20  08
cbb0 : 6a c6 f0 f5 20 c0 cb 9d  eb
cbb8 : cc 03 bd 3c 03 9d 6c 03  d2
cbc0 : 20 ca c2 a0 0f c9 2a d0  94
cbc8 : 02 a0 00 20 af c2 9d 3c  1e
cbd0 : 03 98 9d 9c 03 60 85 b4  cd
cbd8 : 4a 4a 4a 4a 59 6c 03 39  9b
cbe0 : cc 03 29 0f d0 0a a5 b4  b7
cbe8 : 59 3c 03 39 9c 03 [illegible] 0f  ec
cbf0 : 60 68 68 20 cf ff c9 57  75
cbf8 : d0 03 4c 56 cd c9 42 d0  fe
cc00 : 03 4c d0 cd c9 51 d0 03  87
cc08 : 4c 4f cd c9 53 f0 03 4c  0a
cc10 : d1 c2 20 8d c2 48 20 8d  06
cc18 : c2 48 20 49 c2 a0 00 b1  c4
cc20 : fb 8d bc 02 98 91 fb a9  ab
cc28 : 36 8d 16 03 a9 cc 8d 17  70
cc30 : 03 a2 fc 4c ec c3 a2 03  ca
cc38 : 68 9d aa 02 ca 10 f9 68  40
cc40 : 68 ba 8e ae 02 ad a8 02  b3
cc48 : 85 fc ad a9 02 85 fb ad  83
cc50 : bc 02 a0 00 91 fb a9 14  fd
cc58 : 8d 16 03 a9 c2 8d 17 03  e1
cc60 : a9 52 4c ff c2 20 51 c3  3f
cc68 : ad 11 d0 09 10 8d 11 d0  46
cc70 : 60 8d ab 02 08 68 29 ef  0a
cc78 : 8d aa 02 8e ac 02 8c ad  15
cc80 : 02 68 18 69 01 8d a9 02  11
cc88 : 68 69 00 8d a8 02 a9 80  99
cc90 : 8d bc 02 d0 10 20 e5 cd  4b
cc98 : 20 dd fd d8 a2 05 68 9d  70
cca0 : a8 02 ca 10 f9 ad 14 03  61
cca8 : 8d bb 02 ad 15 03 8d ba  5e
ccb0 : 02 ba 8e ae 02 58 ad aa  78
ccb8 : 02 29 10 f0 08 20 65 cc  22
ccc0 : a9 52 4c ff c2 2c bc 02  2a
ccc8 : 50 1f 38 ad a9 02 ed bd  49
ccd0 : 02 8d b1 02 ad a8 02 ed  49
ccd8 : be 02 0d b1 02 d0 67 ad  b0
cce0 : bf 02 d0 5f a9 80 8d bc  0f
cce8 : 02 30 12 4e bc 02 90 cd  0a
ccf0 : ae ae 02 9a a9 cc 48 a9  3f
ccf8 : 70 48 4c ba cd 20 65 cc  04
cd00 : a9 a8 85 fb a9 02 85 fc  99
cd08 : 20 4c c3 a0 00 b1 fb 20  11
cd10 : 2a c3 c8 c0 07 f0 09 c0  04
cd18 : 01 f0 f2 20 4c c3 d0 ed  54
cd20 : ad a9 02 ae a8 02 85 fb  a1
cd28 : 86 fc 20 49 c3 20 cb c4  54
cd30 : 20 c7 c5 20 e4 ff f0 fb  b3
cd38 : c9 4a d0 0a a9 01 8d bc  ee
cd40 : 02 d0 2f ce bf 02 a5 91  16
cd48 : c9 7f d0 26 4c bd cc 20  f0
cd50 : f2 cd a9 40 d0 0a 20 f2  5f
cd58 : cd 08 68 8d aa 02 a9 80  57
cd60 : 8d bc 02 ba 8e ae 02 20  ca
cd68 : 49 c2 20 65 cc ad bc 02  f8
cd70 : f0 37 a2 00 ad 11 d0 a8  9c
cd78 : 29 10 f0 10 98 29 ef 8d  95
cd80 : 11 d0 ea ea a0 0c ca d0  48
cd88 : fd 88 d0 fa 78 a9 47 8d  6a
cd90 : 04 dc 8e 05 dc ad 0e dc  74
cd98 : 29 80 09 11 8d 0e dc a9  76
cda0 : 95 a2 cc 8d bb 02 8e ba  e7
cda8 : 02 ae ae 02 9a 78 ad bb  89
cdb0 : 02 ae ba 02 8d 14 03 8e  9b
cdb8 : 15 03 ad a8 02 48 ad a9  3c
cdc0 : 02 48 ad aa 02 48 ad ab  17
cdc8 : 02 ae ac 02 ac ad 02 40  4d
cdd0 : 20 8d c2 8d be 02 20 8d  b1
cdd8 : c2 8d bd 02 20 8d c2 8d  a5
cde0 : bf 02 4c d6 c2 ad b8 02  0f
cde8 : ae b9 02 8d 14 03 8e 15  63
cdf0 : 03 60 ad 14 03 ae 15 03  11
cdf8 : 8d b8 02 8e b9 02 a9 95  b1
ce00 : 8d 16 03 a9 cc 8d 17 03  2a
ce08 : 60 a9 07 8d 20 d0 a9 36  4c
ce10 : 85 01 a2 00 bd e4 cf 9d  3c
ce18 : d0 02 e8 e0 0d 90 f5 a2  b2
ce20 : 2a 20 40 c3 20 cf ff c9  f7
ce28 : 2a f0 f9 a2 06 dd d2 cf  d7
ce30 : d0 11 8e c1 02 8a 0a aa  57
ce38 : e8 bd d8 cf 48 ca bd d8  b2
ce40 : cf 48 60 ca 10 e7 4c 1f  54
ce48 : ce a9 00 85 fb a9 bf 85  b3
ce50 : fc 85 fe a5 fb 69 04 85  a9
ce58 : fd 20 fc c3 20 e1 ff f0  10
ce60 : 0f ad 8d 02 f0 f6 a9 00  57
ce68 : 85 c6 a5 fc c9 c0 90 e3  06
ce70 : 4c 1f ce 20 7e c2 a0 20  c4
ce78 : a2 00 20 ca c2 20 9a c2  99
ce80 : 81 fb 20 39 c4 d0 f3 20  11
ce88 : 51 c3 4c 24 ce 20 55 cf  35
ce90 : ad c1 02 c9 02 d0 03 4c  23
ce98 : eb ce a2 00 bd 00 bf 8d  89
cea0 : c3 02 e8 bd 00 bf 8d c4  14
cea8 : 02 8a 4c cb ce 20 c2 c2  fa
ceb0 : d0 03 4c 8d ce 20 8d c2  70
ceb8 : 8d c3 02 20 8d c2 8d c4  5a
cec0 : 02 20 55 cf ad c1 02 c9  a6
cec8 : 02 f0 20 20 0d cf a2 0d  42
ced0 : 20 c6 ff a0 00 20 cf ff  a7
ced8 : ea ea ea ea 99 00 bf c8  79
cee0 : d0 f3 20 cc ff 20 bc cf  df
cee8 : 4c 49 ce 20 40 cf a2 0d  b8
cef0 : 20 c9 ff a0 00 b9 00 bf  56
cef8 : 20 d2 ff a6 90 d0 03 c8  83
cf00 : d0 f3 20 cc ff a9 32 20  c2
cf08 : 0d cf 4c b6 cf 8d d1 02  9b
cf10 : ad c3 02 20 79 cf 8e d8  25
cf18 : 02 8d d9 02 ad c4 02 20  e1
cf20 : 79 cf 8e db 02 8d dc 02  a4
cf28 : a2 0f 20 c9 ff a2 00 bd  24
cf30 : d0 02 20 d2 ff e8 e0 0d  49
cf38 : 90 f5 20 cc ff 4c 8c cf  99
cf40 : a2 0f 20 c9 ff a2 00 bd  3c
cf48 : f2 cf 20 d2 ff e8 e0 08  5f
cf50 : 90 f5 4c cc ff a9 0f a8  62
cf58 : a2 08 20 ba ff a9 00 20  eb
cf60 : bd ff 20 c0 ff a9 0d a8  10
cf68 : a2 08 20 ba ff a9 01 a2  04
cf70 : f1 a0 cf 20 bd ff 4c c0  38
cf78 : ff a2 30 38 e9 0a 90 03  13
cf80 : e8 b0 f9 69 3a 60 20 8c  ac
cf88 : cf 4c b6 cf a9 00 85 90  f7
cf90 : 20 51 c3 a9 08 20 b4 ff  d3
cf98 : a9 6f 20 96 ff 20 a5 ff  6b
cfa0 : c9 30 d0 06 4c ab ff 20  d9
cfa8 : a5 ff 20 d2 ff c9 0d d0  d3
cfb0 : f6 20 ab ff 68 68 20 bc  65
cfb8 : cf 4c 1f ce a9 0d 20 c3  5a
cfc0 : ff a9 0f 4c c3 ff a9 06  d0
cfc8 : 8d 20 d0 a9 37 85 01 4c  0b
cfd0 : d6 c2 3a 52 57 4d 58 40  a2
cfd8 : 72 ce ac ce ac ce 48 ce  b6
cfe0 : c5 cf 85 cf 55 31 3a 31  12
cfe8 : 33 20 30 20 31 38 20 30  f1
cff0 : 30 23 42 2d 50 20 31 33  19
cff8 : 20 30 b7 00 ff 00 ff 00  1e
```

Listing 1. »SMON-komplett«-Hauptprogramm. Bitte beachten Sie die Eingabehinweise auf Seite 110.

```
programm : ndisass          ce09 cf3d
-------------------------------------

ce09 : 2b 4b 6b 8b 9b ab bb cb  c4
ce11 : eb 89 93 9f 0b 9c 9e 4e  46
ce19 : 53 52 53 52 53 4c 44 49  f0
ce21 : 43 4f 4c 4c 52 52 41 41  e8
ce29 : 43 53 52 50 4f 41 45 41  4b
ce31 : 58 58 50 43 41 25 26 20  48
ce39 : 21 82 80 81 22 21 82 81  24
ce41 : 03 13 07 17 1b 0f 1f 97  48
ce49 : d7 bf df 02 02 02 02 03  76
ce51 : 03 03 02 02 03 03 a2 02  6e
ce59 : d0 28 a6 ad d0 2b a2 01  90
ce61 : b1 fb c9 9c f0 38 c9 80  0f
ce69 : f0 ec c9 89 f0 e8 29 0f  8c
ce71 : c9 02 f0 16 c9 0a f0 0a  ff
ce79 : e8 c9 04 f0 05 e8 c9 0c  3c
ce81 : d0 1c 86 b6 a2 01 8e c5  d0
ce89 : 02 60 b1 fb 29 90 49 80  e4
ce91 : d0 04 a2 02 d0 ec 86 b6  48
ce99 : a2 0a 8e c5 02 60 a0 02  46
cea1 : 84 b6 a0 00 8c c5 02 b1  0b
cea9 : fb a2 0f dd 08 ce f0 d9  e3
ceb1 : ca d0 f8 29 01 f0 d2 b1  8d
ceb9 : fb 4a 4a 4a 4a 4a 18 69  df
cec1 : 02 8d c5 02 a2 0b b1 fb  7d
cec9 : 3d 40 ce dd 40 ce f0 03  da
ced1 : ca d0 f3 bd 35 ce 85 ab  ef
ced9 : bd 4b ce 85 b6 60 a0 00  91
cee1 : a6 ad f0 06 20 4c c3 4c  67
cee9 : da c5 ae c5 02 d0 06 20  09
cef1 : 4c c3 4c c9 c5 a9 2a 20  fe
cef9 : d2 ff bd 17 ce 20 d2 ff  56
cf01 : bd 21 ce 20 d2 ff bd 2b  81
cf09 : ce 4c 16 c6 a9 00 8d 6b  03
cf11 : c0 8d 6c c0 a9 4c 8d 29  50
cf19 : c5 8d be c5 a9 20 8d 30  3f
cf21 : cd a9 5b 8d 2a c5 a9 ce  60
cf29 : 8d 2b c5 a9 df 8d bf c5  e7
cf31 : 8d 31 cd a9 ce 8d c0 c5  e7
cf39 : 8d 32 cd 00 4c 8c cf a2  00
```

Listing 2. Mit dieser Erweiterung lassen sich illegale Opcodes disassemblieren.

```
programm : floppymon m&t cbf1 cdcb
-------------------------------------

cbf1 : a9 36 85 01 a2 00 bd 90  79
cbf9 : cd f0 06 20 d2 ff e8 d0  36
cc01 : f5 20 48 c3 a2 3e 20 37  9c
cc09 : c3 20 cf ff c9 3e f0 f9  16
cc11 : c9 20 f0 f5 a2 05 dd a6  fc
cc19 : cd f0 09 ca d0 f8 20 48  e0
cc21 : c3 4c 05 cc 8a 0a aa e8  5b
cc29 : bd aa cd 48 ca bd aa cd  99
cc31 : 48 60 20 b9 c2 d0 0a a9  17
cc39 : 00 8d 9e cd 8d 9f cd f0  50
cc41 : 1a 20 84 c2 8d 9f cd 20  32
cc49 : b9 c2 d0 07 a9 00 8d 9e  86
cc51 : cd f0 08 20 84 c2 29 f8  91
cc59 : 8d 9e cd 20 77 cd a2 0f  3b
cc61 : 20 c9 ff a2 00 bd 9b cd  b2
cc69 : 20 d2 ff e8 e0 06 90 f5  7c
cc71 : 20 cc ff a2 0f 20 c6 ff  59
cc79 : a0 00 20 cf ff 99 00 bf  68
cc81 : c8 d0 f7 20 cc ff 4c c4  3b
cc89 : cf a9 bf 85 fc a9 00 85  f5
cc91 : fb 60 20 33 cc 20 8a cc  bc
cc99 : a2 3a 20 37 c3 ad 9f cd  0b
cca1 : 20 21 c3 ad 9e cd 20 21  13
cca9 : c3 a0 20 a2 00 20 43 c3  ae
ccb1 : 20 43 c3 a1 fb 20 21 c3  65
ccb9 : a1 fb 20 20 c4 d0 f1 a9  52
ccc1 : 08 18 6d 9e cd 8d 9e cd  64
ccc9 : 08 c9 f8 d0 06 20 5c cc  7a
ccd1 : 20 8a cc 28 90 09 ee 9f  bb
ccd9 : cd 20 5c cc 20 8a cc 20  31
cce1 : 87 cd 20 e1 ff d0 b1 4c  79
cce9 : 02 cc 20 75 c2 a5 fb 8d  6c
ccf1 : a4 cd a5 fc 8d a5 cd 20  02
ccf9 : 19 cd a0 20 a2 00 20 c1  53
cd01 : c2 20 c1 c2 20 91 c2 20  76
cd09 : d2 ff 20 20 c4 d0 f2 20  c6
cd11 : cc ff 20 cc ff 4c 07 cc  93
cd19 : 20 77 cd a2 0f 20 c9 ff  d6
```

Listing 3. Komfortabler Disketten-Monitor. Bitte beachten Sie die Eingabehinweise auf Seite 110. (Fortsetzung)

```
cd21 : a2 00 bd a1 cd 20 d2 ff  90
cd29 : e8 e0 06 90 f5 60 20 75  e3
cd31 : c2 a2 fd 20 77 c2 a5 fd  e8
cd39 : 8d a4 cd a5 fe 8d a5 cd  cf
cd41 : a9 20 8d a6 cd 20 19 cd  10
cd49 : a0 00 b1 fb 20 d2 ff c8  ff
cd51 : c0 20 90 f6 18 a9 20 6d  4e
cd59 : a4 cd b0 0c 8d a4 cd a9  1a
cd61 : 20 65 fb 85 fb 4c 46 cd  ba
cd69 : 20 cc ff 20 c4 cf a9 08  75
cd71 : 8d a6 cd 4c 02 cc a9 0f  9a
cd79 : a8 a2 08 20 ba ff a9 00  cb
cd81 : 20 bd ff 4c c0 ff 20 e4  60
cd89 : ff f0 fb 60 4c fe cd 0d  19
cd91 : 3e 46 4c 4f 50 2d 4d 4f  32
cd99 : 4e 00 4d 2d 52 00 00 ff  05
cda1 : 4d 2d 57 00 00 08 3a 4d  1f
cda9 : 56 58 40 ea cc 92 cc 2e  8a
cdb1 : cd 8c cd 76 cf a9 00 8d  6c
cdb9 : 22 c0 a9 46 8d df cf a9  d9
cdc1 : cb 8d eb cf a9 f0 8d ea  76
cdc9 : cf 00 56 40 58 ea cc 92  6b
```

Listing 3. (Schluß)

```
programm : ndisass m&t  ce09 cf3d
-------------------------------------

ce09 : 2b 4b 6b 8b 9b ab bb cb  c4
ce11 : eb 89 93 9f 0b 9c 9e 4e  46
ce19 : 53 52 53 52 53 4c 44 49  f0
ce21 : 43 4f 4c 4c 52 52 41 41  e8
ce29 : 43 53 52 50 4f 41 45 41  4b
ce31 : 58 58 50 43 41 25 26 20  48
ce39 : 21 82 80 81 22 21 82 81  24
ce41 : 03 13 07 17 1b 0f 1f 97  48
ce49 : d7 bf df 02 02 02 02 03  76
ce51 : 03 03 02 02 03 03 a2 02  6e
ce59 : d0 28 a6 ad d0 2b a2 01  90
ce61 : b1 fb c9 9c f0 38 c9 80  0f
ce69 : f0 ec c9 89 f0 e8 29 0f  8c
ce71 : c9 02 f0 16 c9 0a f0 0a  ff
ce79 : e8 c9 04 f0 05 e8 c9 0c  3c
ce81 : d0 1c 86 b6 a2 01 8e c5  d0
ce89 : 02 60 b1 fb 29 90 [illegible] 80  e4
ce91 : d0 04 a2 02 d0 ec 86 b6  48
ce99 : a2 0a 8e c5 02 60 a0 02  46
cea1 : 84 b6 a0 00 8c c5 02 b1  0b
cea9 : fb a2 0f dd 08 ce f0 d9  e3
ceb1 : ca d0 f8 29 01 f0 d2 b1  8d
ceb9 : fb 4a 4a 4a 4a 4a 18 69  df
cec1 : 02 8d c5 02 a2 0b b1 fb  7d
cec9 : 3d 40 ce dd 40 ce f0 03  da
ced1 : ca d0 f3 bd 35 ce 85 ab  ef
ced9 : bd 4b ce 85 b6 60 a0 00  91
cee1 : a6 ad f0 06 20 43 c3 4c  1e
cee9 : be c5 ae c5 02 d0 06 20  ed
cef1 : 43 c3 4c ad c5 a9 2a 20  71
cef9 : d2 ff bd 17 ce 20 d2 ff  56
cf01 : bd 21 ce 20 d2 ff bd 2b  81
cf09 : ce 4c f8 c5 a9 00 8d 60  86
cf11 : c0 8d 61 c0 a9 4c 8d 10  5c
cf19 : c5 8d a2 c5 a9 20 8d 28  28
cf21 : cd a9 5b 8d 11 c5 a9 ce  cf
cf29 : 8d 12 c5 a9 df 8d a3 c5  ea
cf31 : 8d 29 cd a9 ce 8d a4 c5  73
cf39 : 8d 2a cd 00 4c 8c cf a2  fc
```

Listing 4. Illegale Opcodes disassemblieren mit der M&T-Version des SMON

```
programm : floppymon    cbf1 cded
-------------------------------------

cbf1 : a9 36 85 01 a2 00 bd b2  bd
cbf9 : cd f0 06 20 d2 ff e8 d0  36
cc01 : f5 20 51 c3 a2 3e 20 40  f0
cc09 : c3 20 cf ff c9 3e f0 f9  16
cc11 : c9 20 f0 f5 a2 05 dd c8  40
cc19 : cd f0 09 ca d0 f8 20 51  f2
cc21 : c3 4c 05 cc 8a 0a aa e8  5b
cc29 : bd cc cd 48 ca bd cc cd  32
cc31 : 48 60 20 c2 c2 d0 0a a9  38
cc39 : 00 8d c0 cd 8d c1 cd f0  ea
cc41 : 1a 20 8d c2 8d c1 cd 20  85
```

Listing 5. Komfortabler Disketten-Monitor für die M&T-Version (Fortsetzung)

```
cc49 : c2 c2 d0 07 a9 00 8d c0  d4
cc51 : cd f0 08 20 8d c2 29 f8  22
cc59 : 8d c0 cd 20 77 cd a2 0f  4c
cc61 : 20 c9 ff a2 00 bd bd cd  3a
cc69 : 20 d2 ff e8 e0 06 90 f5  7c
cc71 : 20 cc ff a2 0f 20 c6 ff  59
cc79 : a0 00 20 cf ff 99 00 bf  68
cc81 : c8 d0 f7 20 cc ff 4c bc  2b
cc89 : cf a9 bf 85 fc a9 00 85  f5
cc91 : fb 60 20 33 cc 20 8a cc  bc
cc99 : a2 3a 20 40 c3 ad c1 cd  b5
cca1 : 20 2a c3 ad c0 cd 20 2a  cc
cca9 : c3 a0 20 a2 00 20 4c c3  d2
ccb1 : 20 4c c3 a1 fb 20 2a c3  0d
ccb9 : a1 fb 20 39 c4 d0 f1 a9  75
ccc1 : 08 18 6d c0 cd 8d c0 cd  31
ccc9 : 08 c9 f8 d0 06 20 5c cc  7a
ccd1 : 20 8a cc 28 90 09 ee c1  ff
ccd9 : cd 20 5c cc 20 8a cc 20  31
cce1 : 87 cd 20 e1 ff d0 b1 4c  79
cce9 : 02 cc 20 7e c2 a5 fb 8d  8e
ccf1 : c6 cd a5 fc 8d c7 cd 20  35
ccf9 : 19 cd a0 20 a2 00 20 ca  65
cd01 : c2 20 ca c2 20 9a c2 20  00
cd09 : d2 ff 20 39 c4 d0 f2 20  e9
cd11 : cc ff 20 bc cf 4c 07 cc  92
cd19 : 20 77 cd a2 0f 20 c9 ff  d6
cd21 : a2 00 bd c3 cd 20 d2 ff  d4
cd29 : e8 e0 06 90 f5 60 20 7e  f5
cd31 : c2 a2 fd 20 80 c2 a5 fd  78
cd39 : 8d c6 cd a5 fe 8d c7 cd  68
cd41 : a9 20 8d c8 cd 20 19 cd  55
cd49 : a0 00 b1 fb 20 d2 ff c8  ff
cd51 : c0 20 90 f6 18 a9 20 6d  4e
cd59 : c6 cd b0 0c 8d c6 cd a9  4d
cd61 : 20 65 fb 85 fb 4c 46 cd  ba
cd69 : 20 cc ff 20 bc cf a9 08  f4
cd71 : 8d c8 cd 4c 02 cc a9 0f  ab
cd79 : a8 a2 08 20 ba ff a9 00  cb
cd81 : 20 bd ff 4c c0 ff 20 e4  60
cd89 : ff f0 fb 60 20 c2 c2 d0  d0
cd91 : 03 4c 86 cf a9 08 20 b1  15
cd99 : ff a9 6f 20 93 ff 20 cf  a6
cda1 : ff 20 a8 ff c9 0d d0 f6  11
cda9 : 20 ae ff 4c 02 cc 4c 09  74
cdb1 : ce 0d 3e 46 4c 4f 50 2d  39
cdb9 : 4d 4f 4e 00 4d 2d 52 00  c9
cdc1 : 00 ff 4d 2d 57 00 00 08  3f
cdc9 : 3a 4d 56 40 58 ea cc 92  7d
cdd1 : cc 2e cd 8c cd ae cd a9  96
cdd9 : 00 8d 22 c0 a9 46 8d d7  f3
cde1 : cf a9 cb 8d e3 cf a9 f0  6f
cde9 : 8d e2 cf 00 03 8e 15 03  da
```

Listing 5. (Schluß)

```
programm : illegal-code 4000 40f4
-------------------------------------

4000 : 87 87 c7 c7 e7 e7 a7 a7  e1
4008 : 27 27 67 67 07 07 47 47  de
4010 : d7 d7 f7 ff 37 aa 17 60  b6
4018 : 57 20 97 13 b7 20 8f 0f  a0
4020 : cf cf 01 8f ef 20 0c af  99
4028 : 19 20 2f 24 30 6f 60 60  62
4030 : 0f 0c 04 4f 20 2a df 05  0d
4038 : 06 ff 0f 04 3f 60 60 7f  fa
4040 : 03 0d 1f 20 23 5f 32 30  ec
4048 : db 60 20 fb 05 11 bf 01  b5
4050 : 04 3b 03 0f 7b 12 33 1b  e0
4058 : 60 01 5b 12 03 83 12 c3  6e
4060 : 32 e3 0c a3 18 23 01 63  61
4068 : 01 03 31 43 31 d3 19 f3  9e
4070 : 60 b3 0c 33 01 73 01 13  e9
4078 : 01 53 32 53 32 ea 6d 54  f3
4080 : 6b 03 81 09 a6 07 b9 02  7c
4088 : 43 27 9d 06 ae 07 a1 1a  65
4090 : 5b 3f 85 1e b6 0f a9 12  5f
4098 : 53 37 8d 16 be 2f 91 2a  ad
40a0 : 6b 0f b5 2e 86 67 99 22  14
40a8 : 63 07 bd 26 8e 67 81 3a  62
40b0 : 7b 1f a5 3e 9b 43 b2 21  cd
40b8 : 57 bd b6 b0 4a b5 20 43  0b
40c0 : ef 36 ce 95 59 ee 49 c4  ec
40c8 : 3f c1 ee 4d e7 3a f0 54  0a
40d0 : 72 d1 00 f7 1e f7 f0 f0  71
40d8 : 5a 7b aa 60 fc f7 c9 60  1e
40e0 : 42 95 59 05 cb f2 ea e4  ae
40e8 : 92 a4 e1 94 a1 c2 03 fb  0b
40f0 : 00 54 20 00 00 00 00 00  23
```

Listing 6. Mit dem Befehl D 4000 erscheinen alle illegalen Opcodes disassembliert auf dem Bildschirm

Neues vom SMON

Diese Erweiterung stellt elf weitere Befehle zur Verfügung. So läßt sich der Monitor zum Beispiel frei im Speicher verschieben und Sprites oder Zeichensätze können sehr einfach erstellt und geändert werden.

Um die Befehlserweiterung zu initialisieren, geht man folgendermaßen vor:

1. SMON absolut laden und NEW eingeben.
2. Den Basic-Lader (Listing 1) eintippen und speichern.
3. Nach dem Start des Laders die Startadresse (dezimal) Ihrer SMON-Version eingeben: zum Beispiel 49152 (= $C000).
4. Den erweiterten SMON zum Beispiel mit "SMONEX" Startadresse Endadresse speichern.

Die neuen Routinen werden, genau wie die meisten bereits vorhandenen, durch einen Buchstaben, zum Teil gefolgt von Adressenangaben, aufgerufen. Bei den ersten drei Ausgabebefehlen kann der Speicherinhalt durch Überschreiben der Zeile geändert werden.

Z 4000 (4100) (Zeichendaten)

gibt den Speicherinhalt von $4000 (bis $40FF) folgendermaßen aus: Jeweils ein Byte pro Zeile wird in 8-Bit-Form dargestellt. Dabei ist ein »*« ein gesetztes, ein ».« dagegen ein nicht gesetztes Bit. Die beiden Zeichen sind willkürlich gewählt und können durch Überschreiben der Speicherzellen $xE65, $xE2D (Bit = 1) und $xE69, $xE30 (Bit = 0) in den Bildschirm-Code (!) der gewünschten Zeichen geändert werden.

Die Anwendung dieses Befehls liegt beispielsweise in der gezielten und anschaulichen Beeinflussung bestimmter Steuerbits in VIC, CIA etc. Andererseits lassen sich — besonders in Verbindung mit dem Kommando »Q« — Zeichendaten leicht modifizieren.

H 4000 (4100)

entspricht dem Befehl »Z« mit dem Unterschied, daß jeweils drei Byte pro Zeile ausgegeben werden. Das entspricht dem Format für Spritedaten. Auf diese Weise steht mit dem erweiterten SMON ein kleiner »Sprite-Editor« zur Verfügung.

N 4000 (4100) (Normaldarstellung)

interpretiert den Speicherinhalt von $4000 (bis $40FF) als Bildschirm-Code und gibt 32 Zeichen pro Zeile aus.

U 4000 (4100) (Übersicht)

Wie »N«, jedoch werden in einer Zeile 40 Zeichen dargestellt. Änderungen sind nur mit »N« möglich. Dieser Befehl dient hauptsächlich dazu, im Speicher abgelegte Bildschirminformationen so auszugeben, wie sie tatsächlich im 40-Zeichen/Zeile-Format aussehen würden. Dieser Befehl ist recht nützlich, um professionelle Videospiele zu analysieren, da hier Spielszenen oft im Bildschirm-Code gespeichert sind.

E 4000 (4100) (Erase)

ist der bereits im 64'er, Ausgabe 2/85 vorgeschlagene Erase-Befehl zum Füllen des Speicherbereiches von $4000 bis $40FF mit $00.

Y 40

kopiert die vorhandene SMON-Version in nur drei Sekunden nach $4000 bis $4FFF und nimmt dabei alle notwendigen Anpassungen vor. Die ursprüngliche Speicherversion des Monitors bleibt unverändert. Mit »G 4000« kann man in den neuen SMON springen. Von dem Byte-Wert, der übergeben werden muß, wird nur das obere Nibble ($4) gewertet, so daß sich theoretisch 16 SMON-Versionen im Speicher unterbringen lassen, wobei natürlich nicht alle Möglichkeiten sinnvoll sind.

Auf diese Weise läßt sich stets die erforderliche Speicherversion herstellen, ohne daß langwierige Änderungen notwendig sind.

Q 2000

kopiert den Zeichensatz aus dem ROM von $D000 bis $DFFF in das RAM nach $2000. Dort kann er mit dem Befehl »Z« nach Belieben geändert werden. Möchte man zum Beispiel das Zeichen »A« in ein »Ä« umdefinieren, so ist der Zeichensatz mit »Q 2000« ins RAM zu kopieren. Anschließend kann mit »Z 2000 2015« der Bereich in binärer Form auf dem Bildschirm ausgegeben werden, in dem auch das Zeichen »A« steht. Dieses kann nun in ein »Ä« geändert werden, indem man mit dem Cursor an die zu ändernde Stelle fährt und für einen Punkt, der gesetzt werden soll, ein »*« und für einen Punkt der nicht gesetzt werden soll ein ».« setzt. So, jetzt ist der Zeichensatz umdefiniert, aber noch nicht aktiviert. Als nächstes muß dem Videocontroller die Startadresse des neuen Zeichensatzes mitgeteilt werden. Dazu ist die Adresse $D018, in der eine hexadezimale 15 steht, durch eine hexadezimale 18 zu ersetzen.

J

bringt den letzten Ausgabebefehl (K, D, M, Z, H, N, U) auf den Bildschirm zurück. Mit RETURN wird der letzte Befehl noch einmal ausgeführt.

Zum Schluß noch ein Tip:

DATA-Zeilen in Hex-Byte-Darstellung sind wegen ihrer konstanten Länge (immer zwei Ziffern pro Wert!) übersichtlicher als solche mit dezimalen Zahlen. Da für die Ausgabe von Hex-Werten bereits alle Routinen im SMON integriert sind, kann der »B«-Befehl (Basic-DATA-Zeilen erzeugen) durch Verändern eines einzigen Sprungbefehles dahingehend manipuliert werden, daß der Speicherinhalt künftig in Form von Hex-Byte ausgegeben wird:

Disassemblieren Sie dazu den Byte-Ausgabebefehl mit »D x99F« und ersetzen »JSR BDD1« durch »JSR x32A«. Für das »x« muß der 4-KByte-Block, in dem die zu ändernde SMON-Version steht, eingesetzt werden. Liegt Ihre SMON-Version bei $C000, so ersetzen Sie das »x« durch ein »C«.

Die Gesamtlänge der DATA-Zeile kann außerdem durch Verändern der Speicherzelle $x9AE variiert werden. Bei dem Wert $1C werden zum Beispiel genau acht Hex-Byte pro Zeile ausgegeben. Das Assembler-Listing zu dieser Erweiterung zeigt Listing 2. (Mark Richters/ah)

```
100 REM ********************************   <238>
110 REM *                              *   <159>
120 REM *      SMON - ERWEITERUNG      *   <222>
130 REM *                              *   <179>
140 REM *  VON   MARK RICHTERS         *   <005>
150 REM *        ALLERSTR.4            *   <068>
160 REM *        2806 OYTEN            *   <037>
170 REM *        TEL.: 04207/1870      *   <078>
180 REM *                              *   <229>
190 REM ********************************   <072>
200 :                                      <176>
210 PRINT"BITTE DIE STARTADRESSE"          <248>
220 PRINT"IHRER SMON-VERSION EINGEBEN!"    <189>
230 INPUT H :AL=H:H=H/4096                 <234>
240 IF H<>INT(H) THEN 210                  <052>
250 POKE AL+38,0                           <239>
260 :                                      <238>
270 DIM W(75)                              <045>
```

Listing 1. Erweiterung zum SMON. Bitte die Eingabe-Hinweise auf Seite 109 beachten.

```
280 FOR I=0 TO 9                              <096>
290 :  W(48+I)=I                              <224>
300 :  W(65+I)=I+10                           <069>
310 NEXT I                                    <140>
320 :                                         <042>
330 FOR I=1 TO 4                              <048>
340 :  READ K,Z : K=K+H*4096                  <123>
350 :  FOR J=K TO K+Z-1                       <235>
360 :    READ A$                              <038>
370 :    A=W(ASC(A$))                         <253>
380 :    B=W(ASC(RIGHT$(A$,1)))               <129>
390 :    S=S+A+B                              <023>
400 :    IF ASC(A$)=42 THEN A=H               <010>
410 :    P=A*16+B : POKE J,P                  <118>
420 :  NEXT J                                 <070>
430 NEXT I                                    <004>
440 :                                         <162>
450 IF S=7031 THEN PRINT"OK!":GOTO 470        <200>
460 PRINT"FEHLER IN DATAS !"                  <071>
470 STOP                                      <026>
480 :                                         <202>
490 DATA 97,7                                 <136>
500 DATA 27,3A,3B,2C,28,29,21                 <025>
510 :                                         <232>
520 DATA 739,28                               <137>
530 DATA A2,06,DD,61,*0,F0,08,CA              <159>
540 DATA 10,F8,A9,2E,20,D2,FF,20              <187>
550 DATA CA,*2,C9,2E,F0,F9,4C,85              <184>
560 DATA*F,4C,B2,*F                           <109>
570 :                                         <038>
580 DATA 781,2                                <220>
590 DATA F0,ED                                <141>
600 :                                         <068>
610 DATA 3593,503                             <202>
620 DATA A9,80,2C,A9,00,85,AB,20              <085>
630 DATA 64,*2,24,AB,10,03,A2,29              <117>
640 DATA 2C,A2,28,20,40,*3,20,23              <239>
650 DATA*3,A0,06,A2,00,A1,FB,0A               <071>
660 DATA 48,B0,03,A9,2E,2C,A9,2A              <144>
670 DATA 91,D1,AD,86,02,91,F3,68              <079>
680 DATA C8,E8,E0,08,D0,E9,20,67              <139>
690 DATA*3,24,AB,30,04,C0,1E,90               <233>
700 DATA DA,20,5D,*4,90,C4,60,A0              <133>
710 DATA 08,2C,A0,18,20,7E,*2,20              <100>
720 DATA B8,*2,A2,08,A9,00,85,AA              <236>
730 DATA 20,CA,*2,C9,2E,F0,07,C9              <052>
740 DATA 2A,F0,04,4C,D1,*2,18,26              <202>
750 DATA AA,88,CA,D0,EB,A5,AA,81              <014>
760 DATA FB,C1,FB,D0,EE,20,67,*3              <071>
770 DATA C0,00,D0,D6,60,A9,80,2C              <218>
780 DATA A9,00,85,AB,20,64,*2,20              <243>
790 DATA 51,*3,24,AB,10,0B,A9,21              <138>
800 DATA 20,D2,FF,20,23,*3,A0,08              <033>
810 DATA 2C,A0,00,A2,00,A1,FB,20              <242>
820 DATA 4F,*4,D0,F9,20,5D,*4,90              <059>
830 DATA DE,60,20,7E,*2,A2,00,A0              <137>
840 DATA 08,B1,D1,81,FB,C1,FB,D0              <193>
850 DATA AA,20,56,*4,90,F3,60,20              <193>
860 DATA 8D,*2,29,F0,85,FF,20,3C              <190>
870 DATA*F,20,D6,*9,20,3C,*F,A9               <001>
880 DATA 14,85,FB,A9,02,05,FF,85              <195>
890 DATA FC,A9,D1,85,FD,A9,0F,05              <008>
900 DATA FF,85,FE,20,68,*A,A0,00              <182>
910 DATA A2,0D,BD,F2,*F,05,FF,85              <072>
920 DATA FC,CA,BD,F2,*F,85,FB,B1              <238>
930 DATA FB,29,0F,05,FF,91,FB,CA              <150>
940 DATA 10,E8,A9,2B,85,FB,A9,00              <089>
950 DATA 05,FF,85,FC,A0,35,B1,FB              <082>
960 DATA 29,0F,05,FF,91,FB,88,88              <137>
970 DATA 10,F4,A9,DE,85,FB,A9,0F              <098>
980 DATA 05,FF,85,FC,A0,13,B1,FB              <088>
990 DATA 29,0F,05,FF,91,FB,88,88              <167>
1000 DATA 10,F4,60,A5,FF,85,A9,20             <096>
1010 DATA 43,*F,68,68,29,F0,85,A5             <117>
1020 DATA 18,69,10,85,A7,A9,00,85             <054>
1030 DATA A4,85,A6,85,A8,60,20,7A             <029>
1040 DATA*2,A9,00,4C,C7,*9,20,7E              <250>
1050 DATA*2,A0,00,A9,D0,84,FD,85              <002>
1060 DATA FE,78,A9,03,85,01,A2,10             <164>
1070 DATA B1,FD,91,FB,C8,D0,F9,E6             <125>
1080 DATA FC,E6,FE,CA,D0,F2,A9,27             <124>
1090 DATA 85,01,58,60,48,C9,4A,D0             <178>
1100 DATA 10,A0,27,B9,00,02,91,D1             <224>
1110 DATA 88,10,F8,68,C6,D6,4C,D6             <214>
1120 DATA*2,A0,06,D9,D7,*F,D0,0A              <187>
1130 DATA A0,27,B1,D1,99,00,02,88             <003>
1140 DATA 10,F8,88,10,EE,68,4C,FF             <033>
1150 DATA*2,A2,0A,DD,D0,*F,F0,06              <139>
1160 DATA CA,D0,F8,4C,D1,*2,20,C5             <228>
1170 DATA*F,4C,D6,*2,8A,0A,AA,BD              <009>
1180 DATA DD,*F,48,BD,DC,*F,48,60             <111>
1190 DATA 28,29,21,45,59,51,48,5A             <209>
1200 DATA 4E,55,44,4B,4D,52,*E,4F             <203>
1210 DATA*E,B2,*E,56,*F,C7,*E,5E              <099>
1220 DATA*F,0B,*E,08,*E,85,*E,88              <157>
1230 DATA*E,06,00,87,03,2D,0C,5C              <076>
1240 DATA 0C,F5,0C,A2,0D,04,0E                <224>
```


Listing 1. Erweiterung zum SMON (Schluß)

```
110:    C026                 .OPT P4,00
                           ;
                           ; !SMON-ERWEITERUNG
                           ; ERWEITERT SMON UM FOLGENDE
                           ; BEFEHLE    - H ADR1 (ADR2)
                           ;            - Z ADR1 (ADR2)
                           ;            - N ADR1 (ADR2)
                           ;            - U ADR1 (ADR2)
                           ;            - E ADR1  ADR2
                           ;            - Y BYTE
                           ;            - Q ADR1
                           ;            - J
                           ;
                           ;-------------------------------
                           ;
700:    C026              S          =   $C000       ;BASISADRESSE SMON
                           ;
720:    C026              HCH        =   "("      ;HIDDEN COMMANDS
730:    C026              HCZ        =   ")"
740:    C026              HCN        =   "!"
                           ;
760:    C026              FLAG       =   $AA
770:    C026              TEMP       =   $AB
780:    C026              PCL        =   $FB
790:    C026              PCH        =   $FC
800:    C026              HINIBBLE   =   $FF
                           ;
820:    C026              CMDS       =   S+$02B
830:    C026              GET12ADR   =   S+$264
840:    C026              GET2ADR    =   S+$27A
850:    C026              GETADR1    =   S+$27E
860:    C026              GETBYT     =   S+$28D
870:    C026              SKIPSPAC   =   S+$2B8
880:    C026              GETRET     =   S+$2C2
890:    C026              GETCHRER   =   S+$2CA
900:    C026              ERROR      =   S+$2D1
910:    C026              EXECUTE    =   S+$2D6
920:    C026              CMDSTORE   =   S+$2FF
930:    C026              CHARRET    =   S+$340
940:    C026              RETURN     =   S+$351
950:    C026              HEXOUT     =   S+$323
960:    C026              PCINC      =   S+$367
970:    C026              ASCII4     =   S+$44F
980:    C026              ASCII5     =   S+$456
990:    C026              CONTIN     =   S+$45D
1000:   C026              OCCUPY     =   S+$9C7
                           ;-------------
1011:   C026                         *=   S+38
1012:   C026 00                      .BYTE00        ;ALTEN Z-BEFEHL LOESCHEN
                           ;
1020:   C061                         *=   S+$61
                           ;
1040:   C061 27 3A 3B HCMDTAB        .ASC "':;,"
1050:   C065 28 29 21                .BYTEHCH,HCZ,HCN
                           ;
1070:   C2E3                         *=   S+$2E3
                           ;
1090:   C2E3 A2 06                   LDX  #6          ;ZEILENANFANG
1100:   C2E5 DD 61 C0 F1             CMP  HCMDTAB,X
1110:   C2E8 F0 08                   BEQ  EXEC1
1120:   C2EA CA                      DEX
1130:   C2EB 10 F8                   BPL  F1
1140:   C2ED A9 2E                   LDA  #"."
1150:   C2EF 20 D2 FF                JSR  $FFD2
                           ;
1170:   C2F2 20 CA C2 EXEC1          JSR  GETCHRERR
1180:   C2F5 C9 2E                   CMP  #"."
1190:   C2F7 F0 F9                   BEQ  EXEC1
1200:   C2F9 4C 85 CF                JMP  LINSTORE
1210:   C2FC 4C B2 CF WEITER         JMP  MORECMD
                           ;
```

Listing 2. Assembler-Listing zur SMON-Erweiterung

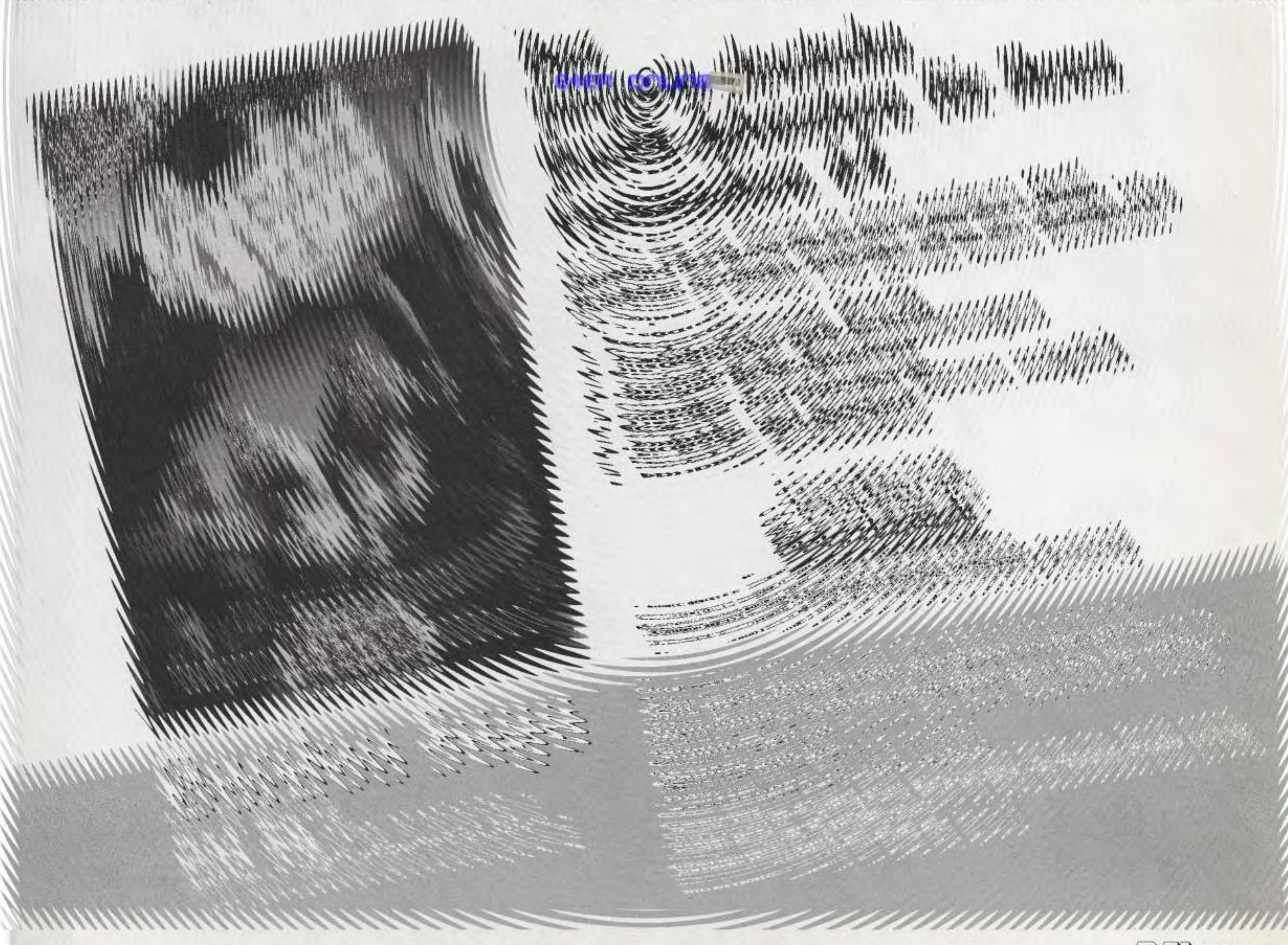
```
1230:  C30D              *=   S+$30D      ;WEITERE BEFEHLE
                      ;
1250:  C30D F0 ED        BEQ  WEITER
                      ;
1270:  CE09              *=   S+$E09      ;NEUE ROUTINEN
                      ;
                      ;-------------
                      ;
1310:  CE09 A9 80  ZCMD   LDA  #$80    ;FLAG SETZEN
1320:  CE0B 2C            .BYTE$2C
                      ;
1340:  CE0C A9 00  HCMD   LDA  #0
1350:  CE0E 85 AB         STA  TEMP
1360:  CE10 20 64 C2      JSR  GET12ADR   ;START/END-ADRESSE
                      ;
1380:  CE13 24 AB  L1     BIT  TEMP
1390:  CE15 10 03         BPL  W8
1400:  CE17 A2 29         LDX  #HCZ       ;HIDDEN COMMAND
1410:  CE19 2C            .BYTE$2C      ;AUSGEBEN
1420:  CE1A A2 28  W8     LDX  #HCH
1430:  CE1C 20 40 C3      JSR  CHARRET
1440:  CE1F 20 23 C3      JSR  HEXOUT     ;PC AUSGEBEN
1450:  CE22 A0 06         LDY  #6         ;SPALTE 6
1470:  CE24 A2 00  L2     LDX  #0
1480:  CE26 A1 FB         LDA  (PCL,X)
                      ;
1500:  CE28 0A     L3     ASL
1510:  CE29 48            PHA           ;BYTE MERKEN
1520:  CE2A B0 03         BCS  BITSET     ;BIT=1, DANN *
1530:  CE2C A9 2E         LDA  #"."       ;BIT=0, DANN .
1540:  CE2E 2C            .BYTE$2C      ;AUSGEBEN
1550:  CE2F A9 2A  BITSET LDA  #"*"
1560:  CE31 91 D1         STA  ($D1),Y
1570:  CE33 AD 86 02      LDA  $286
1580:  CE36 91 F3         STA  ($F3),Y
                      ;
1600:  CE38 68            PLA           ;BYTE ZURUECKHOLEN
1610:  CE39 C8            INY           ;CURSOR WEITER
1620:  CE3A E8            INX           ;NAECHSTES BYTE
1630:  CE3B E0 08         CPX  #8         ;8 BIT
1640:  CE3D D0 E9         BNE  L3         ;BYTE WEITERSCHIEBEN
                      ;
1660:  CE3F 20 67 C3      JSR  PCINC      ;ZAEHLER ERHOEHEN
1670:  CE42 24 AB         BIT  TEMP       ;FLAG FUER 1*8 BIT
1680:  CE44 30 04         BMI  W9         ;ZEILE FERTIG
1690:  CE46 C0 1E         CPY  #30        ;3 BYTES
1700:  CE48 90 DA         BCC  L2
1710:  CE4A 20 5D C4 W9   JSR  CONTIN  ;TASTEN-ABFRAGE
1720:  CE4D 90 C4         BCC  L1
1730:  CE4F 60            RTS
                      ;
                      ;-------------
                      ;
1770:  CE50 A0 08  ZCMDH  LDY  #8
1780:  CE52 2C            .BYTE$2C      ;1 BYTE
                      ;
1800:  CE53 A0 18  HCMDH  LDY  #24     ;3 BYTE
1810:  CE55 20 7E C2      JSR  GETADR1
1820:  CE58 20 B8 C2      JSR  SKIPSPACE  ;SPACES UEBERLESEN
1830:  CE5B A2 08  A1     LDX  #8
1840:  CE5D A9 00         LDA  #0
1850:  CE5F 85 AA         STA  FLAG
                      ;
1870:  CE61 20 CA C2 A2   JSR  GETCHRERR
1880:  CE64 C9 2E         CMP  #"."       ;. => BIT=0
1890:  CE66 F0 07         BEQ  BIT0       ;* => BIT=1
1900:  CE68 C9 2A         CMP  #"*"
1910:  CE6A F0 04         BEQ  BIT1
1920:  CE6C 4C D1 C2 ERR1 JMP  ERROR   ;ANDERES ZEICHEN
                      ;
1940:  CE6F 18     BIT0   CLC
1950:  CE70 26 AA  BIT1   ROL  FLAG
1960:  CE72 88            DEY
1970:  CE73 CA            DEX           ;BYTE
1980:  CE74 D0 EB         BNE  A2         ;NOCH NICHT FERTIG
                      ;
2000:  CE76 A5 AA         LDA  FLAG       ;IN SPEICHER
2010:  CE78 81 FB         STA  (PCL,X)    ;SCHREIBEN
2020:  CE7A C1 FB         CMP  (PCL,X)
2030:  CE7C D0 EE         BNE  ERR1
2040:  CE7E 20 67 C3      JSR  PCINC
                      ;
2060:  CE81 C0 00         CPY  #0
2070:  CE83 D0 D6         BNE  A1         ;NOCH NICHT FERTIG
2080:  CE85 60            RTS
                      ;
                      ;-------------
                      ;
2120:  CE86 A9 80  NCMD   LDA  #$80
2130:  CE88 2C            .BYTE$2C
2140:  CE89 A9 00  UCMD   LDA  #0
2150:  CE8B 85 AB         STA  TEMP
2160:  CE8D 20 64 C2      JSR  GET12ADR
                      ;
2180:  CE90 20 51 C3 L5   JSR  RETURN
2190:  CE93 24 AB         BIT  TEMP
2200:  CE95 10 0B         BPL  U
2210:  CE97 A9 21         LDA  #HCN       ;HIDDEN COMMAND
2220:  CE99 20 D2 FF      JSR  $FFD2
2230:  CE9C 20 23 C3      JSR  HEXOUT
2240:  CE9F A0 08         LDY  #8         ;SPALTE 8
2250:  CEA1 2C            .BYTE$2C
                      ;
2270:  CEA2 A0 00  U      LDY  #0       ;SPALTE 0
2280:  CEA4 A2 00         LDX  #0
                      ;
2300:  CEA6 A1 FB  L4     LDA  (PCL,X)
2310:  CEA8 20 4F C4      JSR  ASCII4     ;ALS BILDSCHIRM-CODE
2320:  CEAB D0 F9         BNE  L4         ;AUSGEBEN
                      ;
2340:  CEAD 20 5D C4      JSR  CONTIN
2350:  CEB0 90 DE         BCC  L5         ;NAECHSTE ZEILE
2360:  CEB2 60            RTS
                      ;-------------
2380:  CEB3 20 7E C2 NCMDH JSR GETADR1
2390:  CEB6 A2 00         LDX  #0
2400:  CEB8 A0 08         LDY  #8         ;SPALTE 8
2410:  CEBA B1 D1  C1     LDA  ($D1),Y
2420:  CEBC 81 FB         STA  (PCL,X)    ;IN SPEICHER
2430:  CEBE C1 FB         CMP  (PCL,X)    ;SCHREIBEN
```

```
2440:  CEC0 D0 AA         BNE  ERR1
2450:  CEC2 20 56 C4      JSR  ASCII5     ;PC ERHOEHEN
2460:  CEC5 90 F3         BCC  C1
2470:  CEC7 60            RTS           ;ZEILE FERTIG
                      ;
                      ;-------------
                      ;
2510:  CEC8 20 8D C2 YCMD JSR  GETBYT
2520:  CECB 29 F0         AND  #%11110000
2530:  CECD 85 FF         STA  HINIBBLE   ;NEUER 4K-BLOCK
2540:  CECF 20 3C CF      JSR  SETPTR     ;ADRESSEN SETZEN
2550:  CED2 20 D6 79      JSR  $79D6      ;W-BEFEHL
                      ;
2570:  CED5 20 3C CF      JSR  SETPTR
2580:  CED8 A9 14         LDA  #$14       ;BEREICH OHNE
2590:  CEDA 85 FB         STA  $FB        ;TABELLEN
2600:  CEDC A9 02         LDA  #2
2610:  CEDE 05 FF         ORA  HINIBBLE
2620:  CEE0 85 FC         STA  $FC
2630:  CEE2 A9 D1         LDA  #<NEWCMDS
2640:  CEE4 85 FD         STA  $FD
2650:  CEE6 A9 0F         LDA  #(NEWCMDS>8 & $F)
2660:  CEE8 05 FF         ORA  HINIBBLE
2670:  CEEA 85 FE         STA  $FE
2680:  CEEC 20 68 7A      JSR  $7A68      ;V-BEFEHL
                      ;
2700:  CEEF A0 00         LDY  #0         ;LADE-BEFEHLE
2710:  CEF1 A2 0D         LDX  #13        ;ANPASSEN
2720:  CEF3 BD F2 CF D3   LDA  CHANGE,X
2730:  CEF6 05 FF         ORA  HINIBBLE
2740:  CEF8 85 FC         STA  PCH        ;ADRESSE ALS ZEIGER
2750:  CEFA CA            DEX
2760:  CEFB BD F2 CF      LDA  CHANGE,X
2770:  CEFE 85 FB         STA  PCL
2780:  CF00 B1 FB         LDA  (PCL),Y
2790:  CF02 29 0F         AND  #%00001111
2800:  CF04 05 FF         ORA  HINIBBLE
2810:  CF06 91 FB         STA  (PCL),Y
2820:  CF08 CA            DEX
2830:  CF09 10 E8         BPL  D3
                      ;
2850:  CF0B A9 2B         LDA  #<CMDS     ;ZEIGER AUF
2860:  CF0D 85 FB         STA  PCL        ;BEFEHLSADRESSEN
2870:  CF0F A9 00         LDA  #(CMDS>8 & $F)
2880:  CF11 05 FF         ORA  HINIBBLE
2890:  CF13 85 FC         STA  PCH
2910:  CF15 A0 35         LDY  #53
2920:  CF17 B1 FB  D1     LDA  (PCL),Y
2930:  CF19 29 0F         AND  #%00001111
2940:  CF1B 05 FF         ORA  HINIBBLE   ;HIBYTES
2950:  CF1D 91 FB         STA  (PCL),Y    ;ANGLEICHEN
2960:  CF1F 88     W1     DEY
2970:  CF20 88            DEY
2980:  CF21 10 F4         BPL  D1
                      ;
3000:  CF23 A9 DE         LDA  #<NEWADR   ;ZEIGER AUF NEUE
3010:  CF25 85 FB         STA  PCL        ;BEFEHLSADRESSEN
3020:  CF27 A9 0F         LDA  #(NEWADR>8 & $F)
3030:  CF29 05 FF         ORA  HINIBBLE
3040:  CF2B 85 FC         STA  PCH
                      ;
3060:  CF2D A0 13         LDY  #TABEND-NEWADR-1
3070:  CF2F B1 FB  D2     LDA  (PCL),Y
3080:  CF31 29 0F         AND  #%00001111
3090:  CF33 05 FF         ORA  HINIBBLE   ;HIBYTES
3100:  CF35 91 FB         STA  (PCL),Y    ;ANGLEICHEN
3110:  CF37 88            DEY
3120:  CF38 88            DEY
3130:  CF39 10 F4         BPL  D2
3140:  CF3B 60            RTS
                      ;
                      ;
3170:  CF3C A5 FF  SETPTR LDA  HINIBBLE
3180:  CF3E 85 A9         STA  $A9        ;NEUER START HI
3190:  CF40 20 43 CF      JSR  GETHI
3200:  CF43 68     GETHI  PLA
3210:  CF44 68            PLA
3220:  CF45 29 F0         AND  #%11110000
3230:  CF47 85 A5         STA  $A5        ;ALTER START HI
3240:  CF49 18            CLC
3250:  CF4A 69 10         ADC  #$10
3260:  CF4C 85 A7         STA  $A7        ;ALTES ENDE HI
                      ;
3280:  CF4E A9 00         LDA  #0         ;LO-BYTES LOESCHEN
3290:  CF50 85 A4         STA  $A4
3300:  CF52 85 A6         STA  $A6
3310:  CF54 85 A8         STA  $A8
3320:  CF56 60            RTS
                      ;
                      ;---------
                      ;
3360:  CF57 20 7A C2 ECMD JSR  GET2ADR
3370:  CF5A A9 00         LDA  #0
3380:  CF5C 4C C7 C9      JMP  OCCUPY
                      ;
                      ;---------
                      ;
3420:  CF5F 20 7E C2 QCMD JSR  GETADR1
3430:  CF62 A0 00         LDY  #0
3440:  CF64 A9 D0         LDA  #$D0       ;ZEIGER AUF
3450:  CF66 84 FD         STY  $FD        ;ZEICHENROM
3460:  CF68 85 FE         STA  $FE
                      ;
3480:  CF6A 78            SEI
3490:  CF6B A9 03         LDA  #%011      ;ROM EINBLENDEN
3500:  CF6D 85 01         STA  1
3510:  CF6F A2 10         LDX  #16        ;4K UEBERTRAGEN
3520:  CF71 B1 FD  E1     LDA  ($FD),Y
3530:  CF73 91 FB         STA  (PCL),Y
3540:  CF75 C8            INY
3550:  CF76 D0 F9         BNE  E1
3560:  CF78 E6 FC         INC  PCH
3570:  CF7A E6 FE         INC  $FE
3580:  CF7C CA            DEX
3590:  CF7D D0 F2         BNE  E1
```

Listing 2. Assembler-Listing zur SMON-Erweiterung (Fortsetzung)

```
3600:  CF7F A9 27             LDA  #$27        ;NORMALEINSTELLUNG
3610:  CF81 85 01             STA  1
3620:  CF83 58                CLI
3630:  CF84 60                RTS
                     ;
                     ;---------
                     ;
3670:  CF85 48       LINSTORE PHA              ;BEFEHL MERKEN
3680:  CF86 C9 4A             CMP  #"J"
3690:  CF88 D0 10             BNE  STORE
                     ;
3710:  CF8A A0 27             LDY  #39
3720:  CF8C B9 00 02 G1       LDA  $0200,Y
3730:  CF8F 91 D1             STA  ($D1),Y      ;ZEILE AUF
3740:  CF91 88                DEY              ;BILDSCHIRM
3750:  CF92 10 F8             BPL  G1           ;SCHREIBEN
3760:  CF94 68                PLA
3770:  CF95 C6 D6             DEC  $D6          ;CURSOR 1 HOCH
3780:  CF97 4C D6 C2          JMP  EXECUTE
                     ;
3800:  CF9A A0 06    STORE    LDY  #6
3810:  CF9C D9 D7 CF G3       CMP  OUTCMDS,Y
3820:  CF9F D0 0A             BNE  W3
                     ;
3840:  CFA1 A0 27    OK1      LDY  #39
3850:  CFA3 B1 D1    G2       LDA  ($D1),Y
3860:  CFA5 99 00 02          STA  $0200,Y      ;ZEILE NUR BEI
3870:  CFA8 88                DEY              ;H,Z,N,U,K,M,D
3880:  CFA9 10 F8             BPL  G2           ;SPEICHERN
                     ;
3900:  CFAB 88       W3       DEY
3910:  CFAC 10 EE             BPL  G3
3920:  CFAE 68       STEND    PLA              ;BEFEHL ZURUECKHOLEN
3930:  CFAF 4C FF C2          JMP  CMDSTORE
                     ;
                     ;+++++++++
                     ;
3970:  CFB2 A2 0A    MORECMD  LDX  #NEWADR-NEWCMDS-3
3980:  CFB4 DD D0 CF B1       CMP  NEWCMDS-1,X
3990:  CFB7 F0 06             BEQ  FOUND
4000:  CFB9 CA                DEX
4010:  CFBA D0 F8             BNE  B1
4020:  CFBC 4C D1 C2          JMP  ERROR
                     ;
4040:  CFBF 20 C5 CF FOUND    JSR  CMDEXEC2
4050:  CFC2 4C D6 C2          JMP  EXECUTE
                     ;
4070:  CFC5 8A       CMDEXEC2 TXA
4080:  CFC6 0A                ASL
4090:  CFC7 AA                TAX
4100:  CFC8 BD DD CF          LDA  NEWADR-1,X
4110:  CFCB 48                PHA
4120:  CFCC BD DC CF          LDA  NEWADR-2,X
4130:  CFCF 48                PHA
4140:  CFD0 60                RTS
                     ;
4160:  CFD1 28 29 21 NEWCMDS  .BYTEHCH,HCZ,HCN
4170:  CFD4 45 59 51          .ASC "EYQ"
4180:  CFD7 48 5A 4E OUTCMDS  .ASC "HZNUDKM"
                     ;
4200:  CFDE 52 CE    NEWADR   .WORDHCMDH-1
4210:  CFE0 4F CE             .WORDZCMDH-1
4220:  CFE2 B2 CE             .WORDNCMDH-1
4230:  CFE4 56 CF             .WORDECMD-1            ;E
4240:  CFE6 C7 CE             .WORDYCMD-1            ;Y
4250:  CFE8 5E CF             .WORDQCMD-1            ;Q
4260:  CFEA 0B CE             .WORDHCMD-1            ;!
4270:  CFEC 08 CE             .WORDZCMD-1            ;Z
4280:  CFEE 85 CE             .WORDNCMD-1            ;N
4290:  CFF0 88 CE             .WORDUCMD-1            ;U
4300:  CFF2          TABEND   =    *
                     ;
4320:  CFF2 06 00    CHANGE   .WORD$0005+1
4330:  CFF4 87 03             .WORD$0386+1
4340:  CFF6 2D 0C             .WORD$0C2C+1
4350:  CFF8 5C 0C             .WORD$0C5B+1
4360:  CFFA F5 0C             .WORD$0CF4+1
4370:  CFFC A2 0D             .WORD$0DA1+1
4380:  CFFE 04 0E             .WORD$0E03+1
```

Listing 2. Assembler-Listing zur SMON-Erweiterung (Schluß)



Schleifen mit Format

In manchen Pascal-Versionen geht es automatisch, beim C 64 nur mit einem Trick: das Einrücken von Schleifen. Anhand eines Programms in Maschinensprache bringen Sie Form in Ihre Listings. Wir zeigen auch, wie es geht.

Anhänger der strukturierten Programmierung haben es gelernt, selbst der in Schulungen: ein Programm modular aufbauen, in kleine Schritte zerlegen, übersichtlich gestalten. Leider ist es mit dem C 64 nicht so ganz einfach. Außer IF..THEN, GOTO, GOSUB und FOR..NEXT wird strukturiertes Codieren nicht unterstützt.

Eine Möglichkeit ist, FOR..NEXT-Schleifen einzurücken. Normalerweise macht man das, indem nach der Zeilennummer ein Doppelpunkt gesetzt wird und erst danach die gewünschte Anzahl Leerstellen. Ein kurzes Beispiel:

Schleife ohne einrücken:

```
10 FOR I=1 TO 10
20 A=A+I
30 GOSUB 100
40 FOR J=1 TO 10
50 PRINT I,J,A*J
60 NEXT J
70 NEXT I
```

Schleife mit einrücken durch Doppelpunkte:

```
10 FOR I=1 TO 10
20 : A=A+I
30 : GOSUB 100
40 : FOR J=1 TO 10
50 :     PRINT I,J,A*J
60 : NEXT J
70 NEXT I
```

Das sieht schon viel besser aus. Bei langen Listings werden Sie diese Form schätzen lernen. Allerdings, das Nonplusultra ist es auch nicht. Besser wäre es, das Listing würde so aussehen:

```
10 FOR I=1 TO 10
20    A=A+I 100
30    GOSUB 100
40    FOR J=1 TO 10
50       PRINT I,J,A*J
60    NEXT J
70 NEXT I
```

Um das zu erreichen, müssen wir eine kleine Maschinenroutine schreiben. Aber wie?

Ein Gedanke liegt nah: Warum nicht die normale LIST-Routine des C 64 verwenden? Sie erledigt ja schon einen großen Teil der Aufgabe, nämlich das normale LISTen. Was fehlt, ist nur noch das Einrücken. Zuerst muß also ein dokumentiertes ROM-Listing her. Dort finden wir die LIST-Routine des Basic-Interpreters ab $A69C. Sie geht bis $A740. Soweit, so gut. Aber wie greifen wir in diese Routine ein? Sieht man sich die LIST-Routine etwas genauer an, findet man einen indirekten Sprung ab Adresse $A717 (JMP ($0306)). Das bedeutet, an dieser Stelle springt die LIST-Routine zu der Adresse, die sich aus dem Inhalt der Adressen $0306/$0307 (LO/HI-Byte) ergibt. Man spricht in so einem Fall auch von einem Vektor. Dort kurz spioniert, finden wir in $0306/$0307 die Zahlen $1A und $A7, zusammengesetzt also die Adresse A71A, das heißt genau die Adresse, die dem JMP($0306) folgt. Damit haben wir genau das, was wir suchen, eine Möglichkeit, in die LIST-Routine einzugreifen. Denn $0306 steht im RAM, kann also (von uns) geändert werden.

Im Prinzip brauchen wir also nur folgendes zu machen: Wir schreiben die Anfangsadresse unserer Routine in die Speicherstellen $0306/$0307. Damit springt die Original-LIST-Routine unsere neue Routine an und führt sie aus. In »Fachkreisen« würde man sagen, der LIST-Vektor ist verbogen worden. Am Ende unseres Programms müssen wir noch dafür sorgen, daß die alte LIST-Routine wieder fortgesetzt wird und zwar machen wir das mit JMP $A71A.

Grundsätzlich kennen wir nun also das Wie. In der Zwischenzeit taucht aber noch ein Gedanke auf: Wie soll das Programm gestartet werden? Nach dem Lesen des Assembler-Kurses (Assembler ist keine Alchimie) im 64'er und einigen anderen Artikeln fällt das Stichwort: Interrupt.

Interrupt war für mich immer ein Wort, vor dem ich mich etwas gedrückt habe, aber so schlimm ist es gar nicht. Doch zuerst einmal zur Aufgabe: Die neue LIST-Routine soll mit der Funktionstaste F7 an- und mit F1 ausgeschaltet werden. Um das zu erreichen, muß noch einmal ein Vektor verbogen werden, und zwar der Interrupt-Vektor in Adresse $0314/$0315 auf unser eigenes Programm (siehe Listing 2, Zeilen 290 bis 360). Wir müssen dafür sorgen, daß die Tasten F7 und F1 dauernd abgefragt werden. Das erreicht man, indem die Interrupt-Routine erweitert wird. Auch hier hilft ein kleines Programm (Listing 2, Zeilen 364 bis 372). Im Prinzip soll unsere neue LIST-Routine so ablaufen:

1. Initialisieren des Programms mit SYS adresse (»adresse« können wir selbst festlegen)
2. mit F7 einschalten und mit F1 ausschalten.
3. mit dem ganz normalen LIST-Befehl ein beliebiges Basic-Programm auf dem Bildschirm oder Drucker ausgeben. Auf dem Drucker sollen die Basic-Befehle FOR und NEXT fettgedruckt werden.

Und damit auch zu sehen ist, ob die neue oder die normale Routine aktiviert ist, soll der Rahmen beim Drücken von F7 die Farbe wechseln (Listing 2, Zeile 452/453) und bei F1 ebenso (Listing 2, Zeile 522/523).

Die LIST-Routine

Die eigentliche neue LIST-Routine finden Sie in Listing 2, Zeile 1000 bis 1340. Danach folgen einige Unterprogramme, zum Beispiel Fettdruck ein-/ausschalten und Leerzeichen ausgeben.

Um das Programm zu verstehen, muß man folgendes wissen:

1. Wenn die neue LIST-Routine angesprungen wird, steht im Akku ein Zeichen aus dem Basic-Listing, das wir ausgeben wollen. Das kann jedes Zeichen sein zwischen der Basic-Zeilennummer und dem Ende einer Basic-Zeile (die Zeilennummer selbst wird vorher, von der alten List-Routine selbst, ausgegeben.).

2. Der Akku-Inhalt muß am Ende unserer LIST-Routine wieder an das Original-LIST übergeben werden. Aus diesem Grund wird er sicherheitshalber am Anfang der Routine mit PHA gesichert und am Ende mit PLA zurückgeholt.

3. Jeder Basic-Befehl wird im Speicher als Token abgelegt, eine Abkürzung. Der Befehl FOR hat den Wert $81 und NEXT den Wert $82.

Der Algorithmus ist jetzt nicht mehr schwer zu entwickeln. Da man FOR..NEXT-Schleifen (fast) beliebig schachteln kann, setzen wir einen Zähler ein (im Listing 2 ZAEHLER genannt), der die Anzahl der Verschachtelungen zählt: Bei jedem FOR wird ZAEHLER um 1 erhöht (Zeile 1210 bis

1240, bei jedem NEXT um 1 vermindert (Zeile 1160 bis 1180). Der Inhalt von ZAEHLER bestimmt auch die Anzahl der Leerzeichen, die am Anfang der Zeile, nach der Zeilennummer, ausgegeben werden (Zeile 1190 bis 1207). Der Anfang der Zeile wird durch eine 4 im Y-Register gekennzeichnet. Das hängt mit dem Aufbau einer Basic-Zeile im C 64 zusammen und mit der indirekt-indizierten Adressierung, die die LIST-Routine verwendet (Y=0 und Y=1 sind die Startadresse der Basic-Zeile, Y=2 und Y=3 die Zeilennummer und ab Y=4 folgt der Rest der Basic-Zeile).

Das Unterprogramm BLANKOUT (Zeile 2010 bis 2070 gibt Leerzeichen aus, und zwar so viele, wie in ZAEHLER stehen.

Die Routinen FETTDRUCK und NORMDRUCK schalten bei Ausgabe auf Drucker (mit OPEN 1,4:CMD1:LIST) Fettdruck an und aus. Interessant sind hier vielleicht die Zeilen 1510 bis 1530 oder 1710 bis 1730. Durch die Abfrage der Speicherstelle $9A (dezimal 154) kann die mit CMD gewählte Geräteadresse überprüft werden (4 steht für Drucker). Die Sequenz, die hier gewählt wurde, um Fettschrift

```
100  -;.LI1,4
110  -;.SY1,4
120  -;.OB"NEULIST.OBJ $5,P,W"
130  -;--------------------------------
140  -; PROGRAMM : LIST-ROUTINE VERAENDERT
142  -; NAME: NEULIST.$5000.SRC
144  -; INIT MIT SYS 5*4096
150  -; MIT F7 AN
160  -; MIT F1 AUSSCHALTEN
170  -; RUECKT FOR..NEXT-SCHLEIFEN EIN
180  -; UND FOR..NEXT AUF DRUCKER FETT
190  -;--------------------------------
200  -.EQ ZAEHLER= $FE;ANZAHL FOR-SCHACHTELUNGEN
210  -.EQ CHROUT  = $FFD2;AUSGABE ZEICHEN
212  -.EQ STROUT  = $AB1E;AUSGABE STRING
220  -.EQ IRQVEC  = $0314;VEKTOR AUF IRQ
230  -.EQ IRQ     = $EA31;IRQ
240  -.EQ CTRL    = $028D ;FLAG FUER CTRL
250  -.EQ KEY     = $00CB ;LETZTE TASTE
260  -;--------------------------------
270  -.BA $5000 ;STARTADRESSE
280  -;--------------------------------
290  -;INTERRUPT AUF EIGENE ROUTINE VERBIEGEN
300  -          SEI
310  -          LDA #<(START)
320  -          STA IRQVEC        ;LO-BYTE
330  -          LDA #>(START)
340  -          STA IRQVEC+1      ;HI-BYTE
350  -          CLI
360  -          RTS
362  -;-------------------------------
364  -START     LDA KEY           ;WELCHE TASTE?
366  -F7        CMP #3            ;F7 ?
367  -          BNE F1            ;NEIN
368  -          JSR LISTNEU       ;NEUE LISTROUTINE
369  -F1        CMP #4            ;F1 ?
370  -          BNE OLDIRQ        ;NEIN
371  -          JSR LISTALT       ;ALTE LISTROUTINE
372  -OLDIRQ    JMP IRQ
373  -;
374  -;-------------------------------
380  -;
390  -;       LISTVEKTOR AUF EIGENE LISTROUTINE
395  -;       START BEI LABEL"LIST"
400  -LISTNEU   LDA #<(LIST)
410  -          STA $0306         ;LISTVEKTOR LO
420  -          LDA #>(LIST)      ;HI-BYTE
430  -          STA $0307
440  -          LDA #0
450  -          STA ZAEHLER
452  -          LDA #$F6          ;BLAUER
453  -          STA $D020         ;RAHMEN
460  -          RTS
470  -;
480  -;
485  -;ALTEN LISTVEKTOR (A71A)
486  -;WIEDERHERSTELLEN MIT F1
490  -LISTALT   LDA #$1A
500  -          STA $0306
510  -          LDA #$A7
520  -          STA $0307
522  -          LDA #$FB          ;GRAUER
523  -          STA $D020         ;RAHMEN
530  -          RTS
540  -;********************************
1000 -;.LI1,4
1010 -LIST      PHA
1020 -          JSR NORMDRUCK
1160 -          CMP #$82          ;NEXT?
1170 -          BNE LI1           ;NEIN
1180 -          DEC ZAEHLER       ;N=N-1
1182 -          JSR FETTDRUCK
1190 -LI1       CPY #4            ;ZEILENANFANG?
1200 -          BNE LI2
1205 -          JSR BLANKOUT
1207 -          JSR BLANKOUT
1210 -LI2       CMP #$81          ;FOR ?
1220 -          BNE LIOUT
1222 -          JSR FETTDRUCK
1240 -          INC ZAEHLER       ;N=N+1
1330 -LIOUT     PLA
1340 -          JMP $A71A         ;LIST
1350 -;
1360 -;--------------------------------
1500 -FETTDRUCK PHA
1510 -          LDA $9A           ;CMD = 4?
1520 -          CMP #4
1530 -          BNE FEOUT         ;NEIN
1540 -          TYA
1550 -          PHA
1560 -          LDA #<(FETT)      ;DRUCKER
1570 -          LDY #>(FETT)      ;AUF
1580 -          JSR STROUT        ;FETTDRUCK
1590 -          PLA
1600 -          TAY
1610 -FEOUT     PLA
1620 -          RTS
1630 -FETT      .BY 27,"E",0      ;FETTDRUCK
1640 -;--------------------------------
1700 -NORMDRUCK PHA
1710 -          LDA $9A           ;CMD4?
1720 -          CMP #4
1730 -          BNE NOOUT         ;NEIN
1740 -          TYA
1750 -          PHA
1760 -          LDA #<(NORM)      ;FETTDRUCK
1770 -          LDY #>(NORM)      ;aus
1780 -          JSR STROUT
1790 -          PLA
1800 -          TAY
1810 -NOOUT     PLA
1820 -          RTS
1830 -NORM      .BY 27,"F",0      ;FETTDRUCK AUS
1840 -;--------------------------------
2000 -;--------------------------------
2010 -BLANKOUT  LDX ZAEHLER       ;AUSGABE N BLANKS
2015 -          BPL BL1           ;WENN < 128
2016 -          INC ZAEHLER
2017 -          BMI BLANKOUT      ;WENN>128
2020 -BL1       BEQ BLOUT
2030 -          JSR BLANK         ;LEERZEICHEN
2050 -          DEX
2060 -          JMP BL1
2070 -BLOUT     RTS
2080 -;
2090 -;--------------------------------
2100 -CR        PHA               ; ZEILENVORSCHUB
2110 -          LDA #13           ;RETURN
2120 -          JSR CHROUT
2130 -          PLA
2140 -          RTS
2150 -;------------------------------
2160 -;
2170 -BLANK     PHA               ;LEERZEICHEN
2180 -          LDA #" "
2190 -          JSR CHROUT
2200 -          PLA
2210 -          RTS
2220 -;
2230 -;--------------------------------
9999 -.EN

READY.
```

Listing 2. Der Assembler-Quelltext zu »Schleifen mit Format«

einzuschalten, gilt für Epson-kompatible Drucker (Zeile 1630 und Normalschrift Zeile 1830). Sie können diese Routinen natürlich entfernen, es ist Geschmacksache, die FOR..NEXT-Schleife auch noch fettgedruckt zu sehen (Listing 1).

Hinweise zum Abtippen

Listing 2 ist der Quelltext oder, wie man auch sagt, der Sourcecode der neuen Listroutine. Wenn Sie wollen, können Sie ihn mit Hypra-Ass eingeben. Das hat den Vorteil, daß Sie das Programm weiterentwickeln oder verändern können. Wollen Sie das Programm lediglich benutzen, tippen Sie am besten Listing 3 mit dem MSE ab. Gestartet wird mit SYS 5*4096 oder SYS 20480. Mit F7 ist die neue LIST-Routine aktiv, mit F1 abgeschaltet. Auch während des LISTens kann umgeschaltet werden. Eine korrekte FOR..NEXT - Schleife wird daran erkannt, daß sich FOR und NEXT auf der gleichen Höhe befinden (Listing 1).

Wenn Sie das Programm etwas genauer analysieren, dürfte es nicht schwerfallen, eigene Wünsche zu verwirklichen.

(H.Zwartscholten/gk)

```
10 PRINTI:PRINTA
20 FORI=1TO4:PRINT"LIST TEST  ":NEXTI
30 A=1
40 B=2
50 FORI=1TO2
55 NEXTI
60 D=4:E=5
70 FORI=1TO2
80   FORJ=1TO2
90     I=3
91   NEXTJ
92   A=5
93 NEXTI
100 FORI=1TO20:PRINTI;:NEXTI
110 A=4:B=5
120 FORJ=1TO2
130   X=X+2:Z=A
140   PRINTX;"ALFA=5:3"
150 NEXTJ
1100 FORI=1TO20:PRINTI;:NEXTI
1110 A=4:B=5
1120 FORJ=1TO2
1130   X=X+2:Z=A
1131   FORQ=1TO2
1132     X=X+2:Z=A
1133     PRINTX;"ALFA=5:3"
1134     IFA=3THENB=4:C=5
1136   NEXTQ
1140   PRINTX;"ALFA=5:3"
1150 NEXTJ

READY.
```

Listing 1. Ein-Beispielprogramm

```
PROGRAMM : NEULIST.OBJ $5  5000 50B8
-------------------------------------

5000 : 78 A9 0D 8D 14 03 A9 50  E2
5008 : 8D 15 03 58 60 A5 CB C9  E2
5010 : 03 D0 03 20 20 50 C9 04  F4
5018 : D0 03 20 34 50 4C 31 EA  FA
5020 : A9 44 8D 06 03 A9 50 8D  E9
5028 : 07 03 A9 00 85 FE A9 F6  00
5030 : 8D 20 D0 60 A9 1A 8D 06  BB
5038 : 03 A9 A7 8D 07 03 A9 FB  D3
5040 : 8D 20 D0 60 48 20 7F 50  42
5048 : C9 82 D0 05 C6 FE 20 68  DD
5050 : 50 C0 04 D0 06 20 96 50  78
5058 : 20 96 50 C9 81 D0 05 20  04
5060 : 68 50 E6 FE 68 4C 1A A7  2A
5068 : 48 A5 9A C9 04 D0 0B 98  87
5070 : 48 A9 7C A0 50 20 1E AB  96
5078 : 68 A8 68 60 1B 45 00 48  C7
5080 : A5 9A C9 04 D0 0B 98 48  BE
5088 : A9 93 A0 50 20 1E AB 68  9F
5090 : A8 68 60 1B 46 00 A6 FE  E5
5098 : 10 04 E6 FE 30 F8 F0 07  E0
50A0 : 20 B0 50 CA 4C 9E 50 60  41
50A8 : 48 A9 0D 20 D2 FF 68 60  9C
50B0 : 48 A9 20 20 D2 FF 68 60  68
```

Listing 3. Der Objectcode zu »Schleifen mit Format«. Benutzen Sie zur Eingabe den MSE.

ÜberLISTet

Mit dieser Betriebssystemerweiterung wird das Schreiben von Basic-Programmen zum Vergnügen. Scrolling aufwärts und abwärts sowie einige Zusatzfunktionen machen es möglich.

Dieses Maschinenprogramm (eventuell in Verbindung mit einem Toolkit) macht das Editieren von Basic-Texten fast so komfortabel wie mit einem guten Textverarbeitungssystem. Man kann damit nämlich nicht nur, wie von vielen teuren Basic-Erweiterungen bekannt, das Listing mit den Cursortasten hinauf und hinunterscrollen, sondern auch:

- andere Erweiterungen verwenden
- einzelne Zeilen löschen und einfügen
- eine Zeile bis beziehungsweise ab Cursorposition löschen
- eine ganze Zeile mit Leerzeichen füllen
- einen Zeilenausschnitt beliebig oft an andere Positionen kopieren.

Dies alles geschieht auf einfachen Tastendruck und ohne Absturzgefahr.

Um mit möglichst vielen anderen Programmen zusammenarbeiten zu können, liegt das zirka 1300 Byte lange Programm im Bereich $7A00 - 7F20 (Hexadezimal). Dazu muß das Ende des Basic-Bereichs gegebenenfalls herabgesetzt werden (POKE 56,122:POKE 55,0: NEW). Mit »SYS 31232« wird die Erweiterung in den Interrupt eingehängt und ist dann sofort einsatzbereit. Es wird nur der IRQ-Vektor verändert.

Das Listing läßt sich im Direktmodus mit Hilfe der Cursortasten fließend hoch- und runterscrollen. Die Programmzeilen können dabei natürlich wie gewohnt geändert und mittels der RETURN-Taste übernommen werden. Stößt man jedoch mit dem Cursor gegen den oberen beziehungsweise unteren Bildschirmrand, wird der Inhalt des Bildschirms in die entsprechende Richtung gescrollt und die nächste Zeile ausgegeben.

Das Programm geht dabei so vor: Es sucht die am nächsten zum Cursor stehende Zeilennummer. Steht keine Zahl in der ersten Spalte, wird normal gescrollt (auch nach unten). Ist die gefundene Zahl größer gleich 64000, wird der Cursor auf diese Zeile gesetzt und das Scrolling verhindert. Andernfalls wird die nächsthöhere beziehungsweise nächstniedrigere Zeile aus dem Basic-Listing gesucht, der Bildschirm in die richtige Richtung gescrollt und die gefundene Zeile gelistet.

Die Erweiterung arbeitet nur im Direktmodus, das heißt wenn in einem Programm ein »INPUT«-Befehl oder ähnliches vorkommt, verhält sich der Bildschirm normal.

Zusätzlich bietet dieses Programm noch einige andere Möglichkeiten, die das Editieren erleichtern. Diese Funktionen werden durch gleichzeitiges Drücken der Control-Taste und einer Buchstabentaste aufgerufen. Zwischen zwei Anführungszeichen werden diese »Control-Codes« jedoch normal ausgegeben.

CTRL-i Einfügen (Insert) einer Leerzeile. Der untere Teil des Bildschirms wird nach unten geschoben. Dies dient zur Übersichtlichkeit, wenn eine neue Basic-Zeile eingegeben werden soll, da keine anderen Zeilen überschrieben werden brauchen.

CTRL-d Löschen (Delete) einer Zeile mit Nachrücken des unteren Bildschirmteils. Auch dies dient der Übersichtlichkeit. Soll eine Zeile gänzlich aus dem Listing

gelöscht werden, muß man (wie üblich) die »leere« Zeilennummer und RETURN eingeben.

CTRL-l Füllen einer Zeile (Line) mit Leerzeichen. Nach den drei letztgenannten Funktionen springt der Cursor an den Anfang der Zeile.

CTRL-b löscht die Zeile bis ausschließlich Cursorposition

CTRL-a löscht die Zeile ab einschließlich Cursorposition

CTRL-u setzt Cursor in die linke untere Ecke

CTRL-k Kopieren eines Zeilenausschnitts. Der Bereich wird durch Invertieren angezeigt und kann mit der Cursor-right/left Taste vergrößert oder verkleinert werden. Bei Betätigen der RETURN- oder der STOP-Taste wird der vorher invertierte Bereich gespeichert.

CTRL-w schreibt (Write) vorher markierten Bereich wieder auf den Bildschirm.

Die Erweiterung läßt sich durch Normalisieren des IRQ-Vektors (zum Beispiel über die Tastenkombination RUN/STOP-RESTORE) oder CTRL-X (exit) abschalten.

Starten von »Listmaster«: Tippen Sie das Listing mit dem MSE ab und speichern Sie es auf Disk oder Kassette. Bei Bedarf laden Sie es absolut (das heißt mit ,8,1 oder ,1,1) und geben ein:

POKE 56,122: POKE 55,0:NEW

Dies ist wie gesagt notwendig, um das Ende des Basic-Bereichs herabzusetzen und das Maschinenprogramm vor der Zerstörung durch Basic-Variablen zu schützen.

Noch ein paar Informationen zu »Listmaster«: Er arbeitet mit anderen Interrupt-Programmen zusammen, da die Routine, auf die der IRQ-Vektor ($0314) vor der Initialisierung von »Listmaster« zeigte, als Unterprogramm aufrufen wird. Man darf also keinesfalls zweimal nacheinander »SYS 31232« eingeben. Das würde einen Selbstaufruf und somit Systemabsturz bewirken. Außerdem benötigt »Listmaster« einige Direkteinsprünge ins Betriebssystem, so daß es bei stark geänderten Varianten desselbigen zu Fehlfunktionen kommen kann.

An Speicheradressen verändert »Listmaster« nur einige wenige, die auch vom Standardbasic oder vom normalen Bildschirmeditor gebraucht werden. Es sind dies im Wesentlichen:

$07	;Zwischenspeicher
$14/15	;Zwischenspeicher
$22/23	;Suchzeiger
$5F/60	;Suchzeiger
$62/63	;Zwischenspeicher
$AC/AD	;Zeiger für Scrolling
$AE/AF	;Zeiger für Scrolling
$B0	;Zwischenspeicher
$CC	;Flag für Cursor blinken
$D1/D2	;Zeiger auf aktuelle Zeile
$D9-F2	;Tabelle Bildschirmzeilen
$F3/F4	;Zeiger auf Farbram
$0314/15	;Interruptvektor

Die Daten für den k-Befehl werden unmittelbar hinter dem »Listmaster«-Programm aufbewahrt (maximal 80 Zeichen).

(Matthias Meyer/tr)

```
programm : listmaster        7a00 7ef5
--------------------------------------
7a00 : ad 14 03 8d 30 7a ad 15   e2
7a08 : 03 8d 31 7a 20 a8 7e 58   5f
7a10 : 60 a5 9d 10 1a a5 cc d0   50
7a18 : 16 a0 00 68 99 c6 7e c8   e7
7a20 : c0 06 d0 f7 a0 00 b9 cc   a1
7a28 : 7e 48 c8 c0 06 d0 f7 4c   74
7a30 : 00 00 a5 c6 f0 54 a5 d4   64
7a38 : d0 50 a5 d8 d0 4c ad 77   ca
7a40 : 02 a0 00 d9 d2 7e f0 07   c0
7a48 : c8 c0 0b d0 f6 f0 3b a5   7c
7a50 : d6 48 a5 d3 48 98 48 a9   ec
7a58 : 01 85 cc 20 99 7e 20 b4   ca
7a60 : 7e 68 0a a8 b9 dd 7e 8d   4a
7a68 : 71 7a b9 de 7e 8d 72 7a   74
7a70 : 4c 00 00 a9 00 85 b0 20   21
7a78 : b3 7d 68 85 d3 68 85 d6   f9
7a80 : 20 6c e5 a9 00 85 cc 20   24
7a88 : a8 7e a0 05 b9 c6 7e 48   94
7a90 : 88 10 f9 4c 81 ea a5 d6   dc
7a98 : f0 05 68 68 4c 83 7a 20   3d
7aa0 : b4 e5 20 21 7d 90 cc a0   44
7aa8 : 01 b1 2b f0 c6 a0 03 b1   4c
7ab0 : 2b c5 15 90 09 d0 bc 88   30
7ab8 : b1 2b c5 14 b0 b5 a0 00   2e
7ac0 : b1 2b 85 22 c8 b1 2b 85   7e
7ac8 : 23 a5 2b 85 5f a5 2c 85   18
7ad0 : 60 a0 01 b1 22 f0 26 a0   7a
7ad8 : 03 b1 22 c5 15 90 09 d0   91
7ae0 : 1c 88 b1 22 c5 14 b0 15   db
7ae8 : a5 22 85 5f a5 23 85 60   36
7af0 : a0 00 b1 5f 85 22 c8 b1   d9
7af8 : 5f 85 23 90 d4 a9 00 85   9a
7b00 : b0 20 b3 7d 20 0d 7e 4c   5a
7b08 : 7a 7a a5 d6 c9 18 d0 05   ae
7b10 : 20 21 7d b0 03 4c 7a 7a   a8
7b18 : a0 01 b1 2b f0 f7 20 b4   c3
7b20 : e5 a2 18 a0 00 20 0c e5   6d
7b28 : a6 f1 30 05 a9 0d 20 16   23
7b30 : e7 a5 b0 c9 18 d0 0b 20   c4
7b38 : 13 a6 90 06 20 18 7e 4c   d9
7b40 : 81 7b a5 2b 85 22 a5 2c   a6
7b48 : 85 23 a0 01 b1 22 f0 4a   2c
7b50 : a0 03 b1 22 c5 15 90 0b   80
7b58 : d0 18 88 b1 22 c5 14 90   4e
7b60 : 02 d0 0f a0 00 b1 22 aa   0e
7b68 : c8 b1 22 86 22 85 23 4c   d6
7b70 : 4a 7b a5 22 85 5f a5 23   56
7b78 : 85 60 20 0d 7e a9 00 85   17
7b80 : c6 68 aa 68 8a c9 28 90   eb
7b88 : 02 e9 28 85 d3 a2 18 86   f9
7b90 : d6 20 f0 e9 20 24 ea 4c   57
7b98 : 83 7a a9 11 8d 77 02 a9   d5
7ba0 : 01 85 c6 d0 dc a6 d6 20   ce
7ba8 : ff e9 a9 00 85 c6 4c 8e   e3
7bb0 : 7d a5 d6 d0 03 4c 73 7a   25
7bb8 : c9 18 f0 e9 aa b5 d9 30   27
7bc0 : 04 09 80 95 d9 8a 48 85   3a
7bc8 : b0 20 b3 7d 20 b4 e5 68   35
7bd0 : aa 4c 8e 7d a5 d6 48 aa   7b
7bd8 : c9 18 f0 c9 20 f0 e9 20   94
7be0 : 24 ea a5 d1 a4 d2 85 ac   6d
7be8 : 84 ad a5 f3 a4 f4 85 ae   90
7bf0 : 84 af e8 20 f0 e9 20 24   b1
7bf8 : ea 20 ff 7d e0 18 d0 dc   6e
7c00 : 20 ff e9 68 48 aa b5 da   0e
7c08 : 29 80 a8 b5 d9 29 7f 95   62
7c10 : d9 98 15 d9 95 d9 e8 e0   43
7c18 : 18 d0 eb a9 80 05 f1 85   cb
7c20 : f1 20 b4 e5 68 aa b5 d9   71
7c28 : 09 80 95 d9 4c 8e 7d a0   82
7c30 : 00 a9 20 91 d1 c8 c4 d3   5d
7c38 : d0 f9 a9 00 85 c6 68 68   70
7c40 : 4c 83 7a a4 d3 a9 20 91   af
7c48 : d1 c8 c4 d5 d0 f9 91 d1   30
7c50 : f0 e8 a9 00 85 c6 a2 18   68
7c58 : 4c 8e 7d 68 68 a9 00 85   37
7c60 : c6 a5 d1 48 a5 d2 48 a5   d4
7c68 : d3 48 a6 d6 20 f0 e9 e0   d7
7c70 : 18 f0 03 a9 4f 2c a9 27   42
7c78 : 85 b0 a5 d3 b4 d9 30 03   1a
7c80 : 38 e9 28 85 d3 a8 20 0c   83
7c88 : 7d 98 48 a5 c6 f0 fc 20   40
7c90 : b4 e5 aa 68 a8 e0 03 f0   6e
7c98 : 20 e0 0d f0 1c e0 1d f0   a9
7ca0 : 06 e0 9d f0 09 d0 e2 c4   c8
7ca8 : b0 f0 de c8 d0 d8 c4 d3   2f
7cb0 : f0 d7 20 0c 7d 88 4c 89   76
7cb8 : 7c c8 84 b0 98 38 e5 d3   5a
7cc0 : 8d f3 7e a4 d3 a2 00 20   0e
7cc8 : 0c 7d 9d f4 7e e8 c8 c4   75
7cd0 : b0 d0 f4 68 85 d3 68 85   d6
7cd8 : d2 68 85 d1 a9 00 85 c6   b8
7ce0 : 4c 83 7a ad f3 7e f0 1b   6f
7ce8 : 8d 92 02 a0 00 98 48 b9   ac
7cf0 : f4 7e ae 86 02 20 13 ea   e3
7cf8 : 20 b6 e6 68 a8 c8 cc f3   26
7d00 : 7e d0 ea 68 68 a9 00 85   8d
7d08 : c6 4c 83 7a b1 d1 49 80   f4
7d10 : 91 d1 60 68 68 a9 00 85   8e
7d18 : c6 78 a9 00 85 cc 4c 8a   8a
7d20 : 7a a5 d6 d0 73 a2 ff e8   5b
7d28 : b5 d9 30 06 e0 19 90 f7   a0
7d30 : b0 13 20 f0 e9 a0 00 b1   97
7d38 : d1 c9 30 90 04 c9 3a 90   a5
7d40 : 06 e0 18 d0 e2 18 60 86   54
7d48 : b0 a0 00 84 14 84 15 b1   f6
7d50 : d1 c9 30 90 04 c9 3a 90   bd
7d58 : 02 38 60 e9 2f 85 07 a5   52
7d60 : 15 85 22 c9 19 b0 25 a5   f1
7d68 : 14 0a 26 22 0a 26 22 65   74
7d70 : 14 85 14 a5 22 65 15 85   ad
7d78 : 15 06 14 26 15 a5 14 65   f4
7d80 : 07 85 14 90 02 e6 15 c8   9e
7d88 : c0 06 d0 c3 68 68 68 68   34
7d90 : a0 00 20 0c e5 4c 83 7a   7e
7d98 : a2 19 ca b5 d9 10 fb 20   7e
7da0 : f0 e9 a0 00 b1 d1 c9 30   de
7da8 : 90 04 c9 3a 90 99 8a d0   96
7db0 : e9 f0 92 a2 18 20 f0 e9   24
7db8 : 20 24 ea a5 d1 a4 d2 85   f2
7dc0 : ac 84 ad a5 f3 a4 f4 85   12
7dc8 : ae 84 af ca 20 f0 e9 20   6f
7dd0 : 24 ea 20 ff 7d e4 b0 d0   d5
7dd8 : dc c6 b0 a2 17 b5 d9 29   71
7de0 : 80 a8 b5 da 29 7f 95 da   18
7de8 : 98 15 da 95 da ca e4 b0   6d
7df0 : d0 eb a9 80 e8 15 d9 95   fa
7df8 : d9 20 ff e9 4c 66 e5 a0   ef
7e00 : 27 b1 d1 91 ac b1 f3 91   f2
7e08 : ae 88 10 f5 60 a0 02 b1   34
7e10 : 5f 85 14 c8 b1 5f 85 15   a6
7e18 : a5 14 85 63 a5 15 85 62   73
7e20 : a2 90 38 20 49 bc 20 df   d7
7e28 : bd a9 01 8d 92 02 a0 00   68
7e30 : b9 00 01 f0 06 20 16 e7   d1
7e38 : c8 d0 f5 a9 20 20 16 e7   46
7e40 : a9 00 85 0f a0 04 b1 5f   dc
7e48 : f0 1b c9 ff f0 07 a6 0f   38
7e50 : d0 03 aa 30 15 20 16 e7   cd
7e58 : c9 22 d0 06 a5 0f 49 01   21
7e60 : 85 0f c8 d0 e1 a9 0d 4c   f1
7e68 : 16 e7 38 e9 7f aa c9 4d   cc
7e70 : b0 20 84 49 a0 ff ca f0   92
7e78 : 08 c8 b9 9e a0 10 fa 30   fd
7e80 : f5 c8 b9 9e a0 08 29 7f   09
7e88 : 20 16 e7 28 10 f3 a4 49   78
7e90 : d0 d0 a9 3f 20 16 e7 90   8e
7e98 : c9 78 ad 30 7a 8d 14 03   79
7ea0 : ad 31 7a 8d 15 03 58 60   c2
7ea8 : 78 a9 11 8d 14 03 a9 7a   e0
7eb0 : 8d 15 03 60 a5 cf f0 0c   49
7eb8 : a5 ce ae 87 02 a0 00 84   8f
7ec0 : cf 20 13 ea 60 ff 00 00   c7
7ec8 : 00 00 00 00 7a 72 24 00   92
7ed0 : 00 00 11 91 0c 09 04 02   64
7ed8 : 01 15 0b 17 18 0a 7b 96   f7
7ee0 : 7a a5 7b b1 7b d4 7b 2f   ec
7ee8 : 7c 43 7c 52 7c 5b 7c e3   cc
7ef0 : 7c 13 7d 00 ff 00 00 ff   55
```

Listing. Zur Eingabe des »Listmasters« verwenden Sie bitte den MSE

Drei »Quickies«

Einzeiler lassen sich nicht nur in Basic, sondern auch in Assembler schreiben! Tippen Sie unsere drei »Quickies« ein, und lassen Sie sich überraschen, mit welchen Tricks da gearbeitet wurde.

Die beiden Basic-Programme »Zeichensatz ändern« und »invertierte Grafik« müssen Sie mit dem MSE eingeben. Dies hat zwei Gründe: Erstens einmal lassen sich dadurch Tippfehler leichter ausschalten und zweitens wäre die Eingabe über Basic sehr mühsam. Wenn Sie die »Einzeiler« auf Diskette gespeichert haben, können Sie sie ohne weiteres wie normale Basic-Listings laden, listen und mit »RUN« starten.

Stop!

```
1 fori = 828to839:reada:pokei,a:next:data72,173,141,2,
201,1,240, 249,104,76,202,241
```

Einschalten poke806,60:poke807,3
Ausschalten poke 806,202:poke807,241

In diesem kleinen Maschinenprogramm wird während des Listens eines Programms die Shift-Taste abgefragt und, wenn diese gedrückt ist, gewartet. Sonst wird zur Adresse 61898 (Ausgabe eines Zeichens) gesprungen. Durch die zwei POKEs wird der Ausgabe-Vektor ($F1CA) auf den Kassettenpuffer nach 828 »verbogen«.

```
828 pha        : Akku sichern
829 lda 653    : Flag für SHIFT
832 cmp #1     : gedrückt?
834 beq 829    : ja, dann warten
836 pla        : Akku wiederherstellen
837 jmp 61898  : zur Ausgaberoutine
```

(Ulrich Pasch/tr)

Zeichensatz ändern

Dieser Einzeiler verschiebt den Zeichensatz in den Speicherbereich Hexadezimal $2000 - $3000. Damit können jederzeit beliebige Zeichen geändert werden; zum Beispiel, um Sonderzeichen zu definieren.

```
programm : chr-ram          0801 0857
-------------------------------------

0801 : 4b 08 01 00 99 22 93 b8  fb
0809 : 12 29 33 45 92 41 12 29  f3
0811 : 92 40 12 20 d0 45 92 df  8d
0819 : 12 44 92 a0 12 20 a0 45  35
0821 : 92 da 12 44 92 db 12 20  be
0829 : 92 30 12 45 92 d8 12 44  c1
0831 : 92 d9 20 12 3f 23 29 37  1a
0839 : 45 92 41 12 29 92 58 12  07
0841 : 4d 92 58 12 d0 92 d8 a0  76
0849 : 22 00 55 08 02 00 9e 31  bf
0851 : 30 32 34 00 00 00 cf ff  e7
```

Listing 1a. Das Basic-Programm zum geänderten Zeichensatz geben Sie bitte mit dem MSE (!) ein. Wenn Sie das Programm listen, werden Sie sehen warum!

Zum Aufbau des Programms (Listing 1a und 1b)

Ein PRINT-Befehl schreibt das Maschinenprogramm in den Bildschirmspeicher, wo es dann mit SYS 1024 gestartet wird.

Das Maschinenprogramm setzt zunächst das Interruptflag, um einen Interrupt durch den CIA 1 zu unterbinden. Das ist nötig, da der Computer sonst abstürzen würde, wenn das Zeichen-ROM eingeschaltet wird. Danach wird das Zeichen-ROM eingeschaltet. Jetzt wird nach der nötigen Parameterübergabe (alter Blockanfang, altes Blockende, neues Blockende) die Blockverschieberoutine des Interpreters aufgerufen. Dann wird der Zeichengenerator wieder abgeschaltet und der Zeiger auf den Zeichengenerator entsprechend geändert (Hexadezimal D018, Bit 1-3). Nach dem Löschen des Interruptflags erfolgt der Rücksprung ins Basic.

```
$0400  78         SEI         Interruptflag setzen
$0401  A9 B3      LDA #$B3
$0403  85 01      STA $01     ;Zeichen-ROM ein
$0405  A9 00      LDA #$00
$0407  A0 D0      LDY #$D0
$0409  85 5F      STA $5F     ;Alter Blockanfang Low-Byte
$040B  84 60      STY $60     ;Alter Blockanfang High-Byte
$040D  A0 E0      LDY #$E0
$040F  85 5A      STA $5A     ;Altes Blockende Low-Byte
$0411  84 5B      STY $5B     ;Altes Blockende High-Byte
$0413  AD 30      LDY #$30
$0415  85 58      STA $58     ;Neues Blockende Low-Byte
$0417  84 59      STY $59     ;Neues Blockende High-Byte
$0419  20 BF A3   JSR $A3BF   ;Blockverschieberoutine
$041C  A9 B7      LDA #$B7
$041E  85 01      STA $01     ;Zeichen-ROM aus
$0420  A9 18      LDA #$18
$0422  8D 18 DO   STA $DO18   ;Zeiger für Zeichengenerator
                              auf $2000
$0425  58         CLI         ;Interruptflag löschen
$0426  60         RTS
```

Listing 1b. Der Quelltext zum geänderten Zeichensatz

(Peter Sajonez/tr)

Invertierte Grafik

Dieser Einzeiler (Listing 2) invertiert eine im Speicher stehende Grafik. Gegenüber dem Einzeiler, der in der 64'er, 3/85 veröffentlicht wurde, braucht er zur Ausführung nur den Bruchteil einer Sekunde. Er besteht im wesentlichen aus einer kleinen Maschinenroutine, die mittels einem PRINT-Befehl in den Bildschirm geschrieben und mit SYS 1024 gestartet wird.

(Frank Meister/tr)

```
programm : grafik invers   0801 084e
-------------------------------------

0801 : 39 08 0a 00 99 22 93 12  de
0809 : 29 bf 45 bd 20 92 20 12  56
0811 : 29 92 40 12 45 bb 44 bc  92
0819 : 29 92 c3 12 2a 29 92 40  75
0821 : 12 28 31 bb 92 c4 12 bd  1e
0829 : 51 bb c8 d0 b7 92 12 a6  4a
0831 : bc a4 bc d0 b1 92 22 00  c1
0839 : 4c 08 14 00 97 31 30 35  bd
0841 : 37 2c 39 36 3a 9e 31 30  61
0849 : 32 34 00 00 00 b3 a9 b9  4d
```

Listing 2. Auch die »invertierte Grafik« geben Sie bitte mit dem MSE ein

INPUT mit Pfiff

Einen INPUT-Befehl, der keine Wünsche mehr offenläßt, stellen wir Ihnen mit diesem Programm vor. Anhand des ausführlich dokumentierten Assembler-Listings können Sie zum Beispiel lernen, wie man eigene Befehle in den Interpreter einbaut.

Exinput 64« löst all Ihre Probleme, die Sie jemals mit dem INPUT-Befehl des Basic V2 gehabt haben. Mit dieser kleinen Erweiterung haben Sie ein Werkzeug in der Hand, das alle seitenlangen GET-Eingabeschleifen vergessen läßt. »Exinput 64« beschränkt sich nicht auf 80 Zeichen; Sie haben, wenn Sie wollen, die volle Länge eines Strings (255 Zeichen) zur Verfügung. Das Eingabefeld wird dabei durch eine Linie sichtbar gemacht. Sämtliche Zeichen können übernommen werden - auch Kommata und Doppelpunkte. Natürlich kann man die Eingabe auch auf bestimmte Tasten beschränken.

Eingabehinweis: Bitte geben Sie Listing 1 mit dem MSE ein, und speichern Sie es mit einem beliebigen Namen.

Programmbeschreibung

Die Erweiterung steht im Speicher von $C000 bis $C51D, wird mit »LOAD "Name",8,1« geladen, mit »SYS 49152« gestartet und muß nur nach einem Reset wieder neu initialisiert werden. Die Routine wurde über einen Zeiger aus der Zero-Page des C 64 in das normale Basic eingebunden.

Dies ist der Zeiger für »Basic-Befehlsadresse holen« in den Adressen $308/$309 (dezimal 776/777). Der Zeiger in diesen beiden Speicherstellen wird von $A7E4 auf den Anfang der Befehlserweiterung, in diesem Fall auf $C00B geändert. Dies wird beim Start des Programms durch die vier ersten Assemblerbefehle gemacht. Danach springt das Programm wieder ins Basic zurück und die beiden zusätzlichen »Basic-Befehle« stehen zum sofortigen Gebrauch im Basic-Programm zur Verfügung (siehe auch Kasten).

Die Routine übernimmt das selektieren der neuen Befehlsworte. Wird ein INPUT- oder PRINT-Befehl gefunden, wird auf das Erkennungszeichen der neuen Befehle (>) geprüft. Ist dieses Zeichen vorhanden, so wird in die eigene Routine verzweigt, ansonsten wird die Routine des Basic-Interpreters aufgerufen.

Der INPUT-Befehl sieht auf den ersten Blick vielleicht etwas kompliziert aus, ist es aber nach ein wenig Ausprobieren nicht mehr.

INPUT> Zeile, Spalte, Länge, Farbe, Text, Code, Zeichen, Variable

Erklärung der Parameter

Zeile: 1 bis 25 (oder Variable)
Spalte: 1 bis 40 (oder Variable)
Länge: 1 bis 255 (oder Variable)
Farbe: 0-15 (Farbe des Textes und des Eingabefeldes)
Text: Beliebiger Text zwischen " ", erscheint vor dem Eingabefeld
Code: - 'A' = **A**lle Zeichen bei der Eingabe erlaubt
- 'B' = Nur **B**uchstaben, Leerzeichen, '.' und '-' für Abkürzung und Trennung
- 'Z' = Nur **Z**iffern, Minuszeichen und Dezimalpunkt
Zeichen: Einzelne Ziffern, Buchstaben oder jedes andere Zeichen zwischen " ", das bei der Eingabe erlaubt ist. (< 30 Zeichen) zum Beispiel Menüpunktabfrage zwischen 1 und 4:
INPUT >20,5,**1**,1,"Bitte wählen: ",**Z**,**"1234"**, X$
Variable: Muß eine String-Variable sein.
Einziger Nachteil: Ziffern müssen zum Rechnen mit VAL umgewandelt werden.

Ganz einfach ist der Befehl, weil man verschiedene Parameter einzeln oder zusammen weglassen kann, und zwar »Farbe«, »Text« und »Zeichen«. Dann bleibt zum Beispiel noch übrig:

INPUT>10,3,15,B,A$

Die Länge des Eingabefeldes wird durch eine Linie markiert, dessen einzelnes Zeichen (CHR$(100)) an der

Erklärungen zu oft benötigten ROM-Routinen und zur Erstellung der Variablen.

CHRGET: Diese Routine steht ab $E3A2 im Betriebssystem und wird beim Einschalten des Computers nach $0073 in die Zero-Page kopiert. Der Zeiger in $007A/$007B zeigt immer auf das aktuelle Zeichen im Basic-Text und beim Ansprung der Routine wird jeweils das nächste Zeichen in den Akku geholt.

CHRGOT: Dies ist nur eine andere Einsprungadresse der CHRGET-Routine. Hier wird das aktuelle Zeichen in den Akku geholt und die Flags wieder gesetzt, ohne den Zeiger in $7A/$7B vorher zu erhöhen.

CHKCOM: Hier handelt es sich um eine Basic-Routine, die das laufende Zeichen prüft. Beim Einsprung in »CHKCOM« ($AEFD) wird auf ein Komma im Basic-Text geprüft, bei $AEF7 auf »Klammer zu« und bei $AEFA auf »Klammer auf«. Stimmen die Zeichen nicht überein, wird automatisch ein »Syntax Error« ausgegeben.
Vor dem Rücksprung liest die Routine gleich das nächste Zeichen aus dem Basic-Text in den Akku, so daß dies beim Weiterführen des Programms entfallen kann.

GETBYT: Diese Basic-Routine holt eine Zahl (Byte 0-255) in das X-Register. Beim Einsprung in $B79B wird zuerst durch »JSR $0073« das nächste Zeichen geholt (CHRGET). Hat man vorher jedoch mit »JSR $AEFD« auf ein Komma geprüft (CHKCOM), darf man erst bei $B79E einspringen, weil das nächste Zeichen sich bereits im Akku befindet.

GETVAR: Diese Routine holt sich den Variablennamen aus dem Basic-Text und sucht diese Variable dann im Speicher. Wenn sie nicht vorhanden ist, wird sie angelegt. Danach stehen dann in $0D das Typflag dieser Variablen (String=$FF/Zahl=$00), im Akku das Low-Byte und im Y-Register das High-Byte der Adresse, an der der Stringdescriptor beginnt.

FRESTR: Mit dieser Routine wird die weitere Auswertung der mit »GETVAR« eingelesenen Variablen vorgenommen. Nachher stehen im X-Register das Low-Byte der Adresse, im Y-Register das High-Byte der Adresse und im Akku die Länge des Strings.

STRPOI: Hiermit berechnet man den Stringzeiger und reserviert gleichzeitig Speicher für den String am Ende des Stringspeichers. Die Länge des Strings muß vorher im Akku stehen, wo sie sich nach »FRESTR« ja auch befindet. Man findet dann in $61 die Länge des reservierten Bereichs, in $62 das Low-Byte der Adresse und in $63 das High-Byte der Adresse des reservierten Bereichs.

Adresse 49244 steht und durch »POKE 49244, CHR$-Code« beliebig geändert werden kann, zum Beispiel 32 (Leerzeichen), 46 (Punkt) etc.

Wird eine unerlaubte Taste gedrückt, oder soll der Cursor die Feldgrenze überschreiten, ertönt zur Warnung ein Piepston. Dieser kann mit »POKE 50340,0« ausgeschaltet und mit »POKE 50340,65« wieder eingeschaltet werden.

Am Ende des Eingabefeldes wird das letzte Zeichen bei weiteren Eingaben überschrieben.

Editier-Tasten

CRSR Left	
CRSR Right	
CRSR Up	(Nur bei mehreren Zeilen)
CRSR Down	(Nur bei mehreren Zeilen)
INST	
DEL	
HOME	
CLR/HOME	
RETURN	übernimmt den Feldinhalt ohne Leerstellen am Ende
SHIFT/RETURN	übernimmt auch die Leerstellen, also das ganze Feld

Ob mit oder ohne Leerstellen kann bei den beiden Tasten auch selbst bestimmt werden.
Für RETURN: »POKE 49595, 182(ohne)/208(mit)«
Für SHIFT/RETURN: »POKE 49602, 208(mit)/182(ohne)«

F8	läßt einen eventuellen Inhalt der Variablen im Eingabefeld erscheinen, jedoch nur bis zur Länge des Feldes selbst

Der zweite Befehl der Erweiterung betrifft den PRINT-Befehl.

Die Syntax des PRINT-Befehls

PRINT > Zeile, Spalte, Farbe, String

Hier kann die Farbe weggelassen werden, der zu druckende String muß zwischen " " stehen wie beim normalen PRINT-Befehl und jeder Parameter kann durch eine Variable ersetzt werden.

Mit den Angaben der im Kasten genannten Routinen kann man leicht eine Variable erstellen. Man kopiert nur die Adresse des reservierten Bereichs und die Länge in den Stringdescriptor und den Text der Variablen schreibt man in den reservierten Bereich. Alles andere erfahren Sie, wenn Sie das Programm selbst durchlesen (Listing 2).

(Guy Mootz/og)

```
PROGRAMM : EXINPUT 64        C000 C51D
--------------------------------------

C000 : A9 0B A0 C0 8D 08 03 8C  AD
C008 : 09 03 60 A2 9C 8E 00 D4  E7
C010 : A2 48 8E 01 D4 A2 00 8E  1A
C018 : 02 D4 8E 04 D4 A2 08 8E  48
C020 : 03 D4 A2 0A 8E 05 D4 A2  21
C028 : 64 8E 06 D4 A2 0F 8E 18  FD
C030 : D4 20 73 00 C9 85 F0 0D  98
C038 : C9 99 F0 06 20 79 00 4C  31
C040 : E7 A7 4C 4D C4 20 73 00  D3
C048 : C9 B1 F0 32 20 BF AB 4C  B3
C050 : AE A7 4C 08 AF 00 00 00  E1
C058 : 00 00 00 00 64 00 00 00  9F
C060 : 00 00 00 00 00 00 00 00  61
C068 : 00 00 00 00 00 00 00 00  69
C070 : 00 00 00 00 00 00 00 00  71
C078 : 00 00 00 00 00 00 A2 00  03
C080 : 8E 5B C0 8E 59 C0 20 9B  11
C088 : B7 8A 38 E9 01 C9 19 B0  F4
C090 : 11 8D 56 C0 20 FD AE 20  02
C098 : 9E B7 8A 38 E9 01 C9 28  DA
C0A0 : 90 03 4C 48 B2 A8 8C 57  1F
C0A8 : C0 AE 56 C0 20 0C E5 20  A7
C0B0 : FD AE 20 9E B7 E0 00 F0  45
C0B8 : E9 8E 58 C0 AC 86 02 8C  37
C0C0 : 5A C0 20 FD AE C9 39 B0  C2
C0C8 : 11 C9 2F 90 0D 20 9E B7  57
C0D0 : E0 10 B0 CE 8E 86 02 20  24
C0D8 : FD AE C9 22 F0 03 4C 00  3B
C0E0 : C1 A0 01 B1 7A C9 22 F0  C8
C0E8 : 0A 20 D2 FF C8 4C E3 C0  B7
C0F0 : 4C 99 AD 38 98 65 7A 90  3B
C0F8 : 02 E6 7B 85 7A 20 FD AE  FB
C100 : C9 5A F0 0B C9 42 F0 07  14
C108 : C9 41 F0 03 4C 08 AF 8D  ED
C110 : 59 C0 20 73 00 20 FD AE  96
C118 : A2 00 C9 22 D0 1E 20 73  D6
C120 : 00 C9 22 F0 0E E0 1E D0  AD
C128 : 03 20 71 A5 9D 60 C0 E8  FE
C130 : 4C 1E C1 8E 5B C0 20 73  F1
C138 : 00 20 FD AE 20 8B B0 A6  0C
C140 : 0D F0 AD 8D 5E C0 8C 5F  BF
C148 : C0 A0 00 AD 5C C0 C9 64  CA
C150 : D0 02 A9 A4 20 D2 FF C8  4A
C158 : CC 58 C0 D0 EE A0 00 A9  E2
C160 : 9D 20 D2 FF C8 CC 58 C0  98
C168 : D0 F5 A4 D3 A6 D6 8C 57  D8
C170 : C0 8E 56 C0 20 24 EA A5  3F
C178 : D2 85 FE AC 57 C0 98 18  70
C180 : 65 D1 90 02 E6 FE 85 FD  AB
C188 : A5 F4 85 FC 98 18 65 F3  70
C190 : 90 02 E6 FC 85 FB A9 00  59
C198 : 8D 55 C0 46 CC 46 CF 20  47
C1A0 : E4 FF F0 F7 48 E6 CC A4  F7
C1A8 : D3 B1 D1 29 7F 91 D1 A5  04
C1B0 : 91 C9 7F F0 41 68 C9 0D  BC
C1B8 : D0 03 4C B6 C3 C9 8D D0  56
C1C0 : 03 4C D0 C3 C9 14 F0 37  05
C1C8 : C9 9D F0 33 C9 1D F0 38  BC
C1D0 : C9 94 F0 28 C9 91 F0 2A  66
C1D8 : C9 13 F0 29 C9 8C F0 2B  A7
C1E0 : C9 85 B0 03 4C EF C1 C9  D7
C1E8 : 8C 90 0B C9 93 F0 16 C9  65
C1F0 : 11 D0 1B 4C 3A C3 20 A3  43
C1F8 : C4 4C 9B C1 4C CA C2 4C  C0
C200 : 73 C2 4C 14 C3 4C 75 C3  66
C208 : 4C 4E C2 4C C1 C4 AE 59  65
C210 : C0 E0 41 F0 26 C9 2E F0  FA
C218 : 22 C9 2D F0 1E E0 5A F0  BC
C220 : 0F C9 20 F0 16 C9 41 90  10
C228 : CD C9 5B 90 0E 4C F6 C1  65
C230 : C9 30 90 C2 C9 3A 90 03  44
C238 : 4C F6 C1 AE 5B C0 E0 00  85
C240 : F0 C6 CA DD 60 C0 F0 06  DD
C248 : CA 10 F8 4C F6 C1 AE 55  C5
C250 : C0 EC 58 C0 F0 0B EE 55  82
C258 : C0 AE 55 C0 EC 58 C0 D0  13
C260 : 09 C9 1D F0 0B 20 16 E7  8D
C268 : A9 9D 20 16 E7 4C 9B C1  7D
C270 : 4C F6 C1 AC 55 C0 C0 00  9C
C278 : F0 F6 CC 58 C0 D0 1F C9  C4
C280 : 9D D0 0D AC 58 C0 C0 01  EF
C288 : F0 38 CE 55 C0 4C 9E C2  61
C290 : CE 55 C0 AC 55 C0 AD 5C  99
C298 : C0 91 FD 4C 9B C1 CE 55  D7
C2A0 : C0 C9 9D F0 C5 B1 FB AA  F9
C2A8 : B1 FD 88 91 FD 8A 91 FB  1E
C2B0 : C8 C8 CC 58 C0 90 EE 88  78
C2B8 : AD 5C C0 91 FD A9 9D 4C  32
C2C0 : 6A C2 A0 00 8C 55 C0 4C  C3
C2C8 : F6 C1 AC 58 C0 CE 58 C0  3A
C2D0 : 88 C0 FF F0 33 CC 55 C0  47
C2D8 : 90 2E B1 FD CD 5C C0 F0  50
C2E0 : EF CC 58 C0 F0 22 B1 FB  42
C2E8 : AA B1 FD C8 91 FD 8A 91  5A
C2F0 : FB 88 88 C0 FF F0 05 CC  9F
C2F8 : 55 C0 B0 EA C8 AD 5C C0  24
C300 : 91 FD EE 58 C0 4C 9B C1  B7
C308 : 48 8A 48 20 A3 C4 68 AA  03
C310 : 68 4C 02 C3 AD 55 C0 38  90
C318 : E9 28 90 1B AE 55 C0 EC  0F
C320 : 58 C0 D0 0B 8D 55 C0 A9  48
C328 : 1D 20 16 E7 4C 32 C3 8D  58
C330 : 55 C0 A9 91 4C 6A C2 4C  3E
C338 : F6 C1 AD 58 C0 C9 28 F0  62
C340 : 0F AD 55 C0 18 69 28 B0  62
C348 : 07 CD 58 C0 F0 05 90 0E  FA
C350 : 4C F6 C1 8D 55 C0 A9 9D  77
C358 : 20 16 E7 4C 61 C3 8D 55  1C
C360 : C0 A9 11 20 16 E7 AE 58  49
C368 : C0 CA EC 55 C0 D0 03 EE  F0
C370 : 55 C0 4C 9B C1 48 AC 55  68
C378 : C0 F0 0E CC 58 C0 D0 01  9E
C380 : 88 A9 9D 20 D2 FF 88 D0  39
C388 : F8 68 C9 93 F0 0A C9 FF  20
C390 : F0 03 4C 96 C1 4C C6 C4  0B
C398 : AC 58 C0 AD 5C C0 C9 64  12
C3A0 : D0 02 A9 A4 20 D2 FF 88  1A
C3A8 : D0 F1 A9 13 48 AC 58 C0  0A
C3B0 : 4C B1 C3 4C 96 C1 AC 58  12
C3B8 : C0 4C C0 C3 A9 20 91 FD  25
C3C0 : 88 C0 FF F0 07 B1 FD CD  58
C3C8 : 5C C0 F0 F0 C8 8C 58 C0  B2
C3D0 : AC 58 C0 88 C0 FF F0 0E  D5
C3D8 : B1 FD CD 5C C0 D0 F4 A9  40
C3E0 : 20 91 FD 4C D3 C3 AE 5A  9C
C3E8 : C0 8E 86 02 AE 58 C0 F0  64
C3F0 : 4B 20 A3 B6 AD 5F C0 85  EF
C3F8 : 65 AD 5E C0 85 64 AD 58  C6
C400 : C0 20 75 B4 84 FB A0 00  6F
C408 : 91 64 C8 8A 91 64 C8 A5  F9
C410 : FB 91 64 A0 01 B1 64 48  C1
C418 : C8 B1 64 85 65 68 85 64  FB
C420 : AC 58 C0 88 B1 FD 29 7F  E8
C428 : C9 20 90 19 C9 40 90 06  35
C430 : C9 60 90 15 69 3F 91 64  90
C438 : C0 00 D0 E7 AE 56 C0 20  0A
C440 : 0C E5 4C AE A7 09 40 D0  8D
C448 : ED 09 20 D0 E9 20 73 00  49
C450 : C9 B1 F0 06 20 A0 AA 4C  39
C458 : AE A7 20 9B B7 8A 38 E9  DA
C460 : 01 C9 19 B0 0F 48 20 FD  52
C468 : AE 20 9E B7 8A 38 E9 01  D9
C470 : C9 28 90 03 4C 48 B2 A8  F5
C478 : 68 AA 20 0C E5 AD 86 02  A9
C480 : 8D 5A C0 20 FD AE C9 39  5D
C488 : B0 0D C9 2F 90 09 20 9E  26
C490 : B7 8E 86 02 20 FD AE 20  5D
C498 : A0 AA AD 5A C0 8D 86 02  DB
C4A0 : 4C AE A7 A2 41 8E 04 D4  C4
C4A8 : A2 90 86 02 CA D0 FD C6  2D
C4B0 : 02 F0 05 A2 FF 4C AC C4  5F
C4B8 : A2 00 8E 04 D4 60 4C 96  2D
C4C0 : C1 A9 FF 4C 75 C3 A0 00  D7
C4C8 : B1 47 99 61 00 C8 C0 03  FF
C4D0 : D0 F6 A5 61 CD 58 C0 90  75
C4D8 : 03 AD 58 C0 8D 5D C0 A0  E8
C4E0 : 00 C9 00 F0 D9 B1 62 C9  2B
C4E8 : 20 D0 07 AD 5C C0 91 FD  F6
C4F0 : A9 1D 20 D2 FF C8 18 CC  CA
C4F8 : 5D C0 B0 03 4C E5 C4 AD  A4
C500 : 5C C0 CC 58 C0 B0 06 91  C7
C508 : FD C8 4C 02 C5 AC 5D C0  75
C510 : A9 9D 20 D2 FF 88 C0 00  31
C518 : D0 F6 4C 96 C1 FF 00 FF  65
```

Listing 1. Geben Sie »Exinput 64« bitte mit dem MSE ein

```
                  ASM VERS. 4.23
                  ;----------------------------------------
                  ;-                                      -
                  ;-        EXTENDED INPUT 64             -
                  ;-                                      -
                  ;-           GUY MOOTZ                  -
                  ;-                                      -
                  ;-         BOUVARTSTR. 11               -
                  ;-         L 7519  MERSCH               -
                  ;-           LUXEMBURG                  -
                  ;-                                      -
                  ;-           15.10.85                   -
                  ;-                                      -
                  ;----------------------------------------
                  ;
                  *      =$C000  ;STARTADRESSE DEZ.49152
                  BINPUT =$ABBF  ;BASIC INPUT
                  BPRINT =$AAA0  ;BASIC PRINT
                  BSOUT  =$FFD2  ;GIBT ZEICHEN AUS
                  CHRGET =$0073  ;HOLT ZEICHEN AUS BASIC-T
                  EXT
                  CHRGOT =$0079  ;FLAGS WIEDER SETZEN
                  CURCOL =$0286  ;MOMENTANE ZEICHENFARBE
                  CURSOR =$E50C  ;SETZT CURSOR
                  CKCOM  =$AEFD  ;PRUEFT AUF KOMMA
                  EXESTA =$A7E7  ;STATEMENT AUSFUEHREN
                  FRESTR =$B6A3  ;STRINGVERWALTUNG
                  GETIN  =$FFE4  ;HOLT ZEICHEN IN A
                  GETBYT =$B79B  ;HOLT BYTE(0-255) NACH X
                  GETBY  =$B79E  ;ID.OHNE ZEICHEN ZU HOLEN
                  GETVAR =$B08B  ;VARIABLE HOLEN
                  INTERP =$A7AE  ;INTERPRETERSCHLEıFE
                  ILQUAN =$B248  ;GIBT ILLEGAL QUANT. AUS
                  OUTSCR =$E716  ;ZEICHEN AUF BILDSCHIRM
                  SYNTAX =$AF08  ;GIBT SYNTAX ERROR AUS
                  STRLON =$A571  ;GIBT STR TOO LONG AUS
                  STRPOI =$B475  ;STRINGZEIGER BERECHNEN
                  TYPMIS =$AD99  ;GIBT TYPE MISMATCH AUS
                  ;
                  ;----------------------------------------
                  ;    ZEIGER AUF EIGENE ERWEITERUNG
                  ;----------------------------------------
                  ;
C000 A9 0B               LDA #$0B       ;LB START
C002 A0 C0               LDY #$C0       ;HB START
C004 8D 08 03            STA $0308      ;LB/HB VEKTOR FUER
C007 8C 09 03            STY $0309      ;BASIC-BEFEHLSADR.
C00A 60                  RTS            ;HOLEN
                  ;
                  ;----------------------------------------
                  ;   SID-PARAMETER FUER PIEPTON SETZEN
                  ;----------------------------------------
                  ;
C00B A2 9C               LDX #$9C
C00D 8E 00 D4            STX $D400
C010 A2 48               LDX #$48
C012 8E 01 D4            STX $D401
C015 A2 00               LDX #$00
C017 8E 02 D4            STX $D402
C01A 8E 04 D4            STX $D404
C01D A2 08               LDX #$08
C01F 8E 03 D4            STX $D403
C022 A2 0A               LDX #$0A
C024 8E 05 D4            STX $D405
C027 A2 64               LDX #$64
C029 8E 06 D4            STX $D406
C02C A2 0F               LDX #$0F
C02E 8E 18 D4            STX $D418
                  ;
                  ;----------------------------------------
                  ;        BASIC BEFEHLE PRUEFEN
                  ;----------------------------------------
                  ;
C031 20 73 00            JSR CHRGET     ;ZEICHEN HOLEN
C034 C9 85               CMP #$85       ;INPUT-BEFEHL ?
C036 F0 0D               BEQ INPUT      ;JA,NACH 'INPUT'
C038 C9 99               CMP #$99       ;PRINT-BEFEHL ?
C03A F0 06               BEQ PRINT1     ;JA,NACH 'PRINT1'
C03C 20 79 00            JSR CHRGOT     ;NEIN,FLAGS SETZEN+
C03F 4C E7 A7            JMP EXESTA     ;BEFEHL AUSFUEHREN
C042 4C 4D C4     PRINT1 JMP PRINT
                  ;
                  ;----------------------------------------
                  ;    AUF NEUEN INPUT-BEFEHL PRUEFEN
                  ;----------------------------------------
                  ;
C045 20 73 00     INPUT  JSR CHRGET     ;ZEICHEN HOLEN
C048 C9 B1               CMP #$B1       ;'>' ?
C04A F0 32               BEQ ACCINP     ;JA, NEUEN BEFEHL
C04C 20 BF AB            JSR BINPUT     ;NEIN, BASIC INPUT
C04F 4C AE A7            JMP INTERP     ;
C052 4C 08 AF     SYNOUT JMP SYNTAX     ;
                  ;
                  ;----------------------------------------
                  ; SPEICHERPLATZ FUER BEFEHLSPARAMETER
                  ;----------------------------------------
                  ;
C055 00           COUNT  .BYTE 0        ;ZAEHLER IM FELD
C056 00           CRSLIN .BYTE 0        ;CURSORZEILE
C057 00           CRSCOL .BYTE 0        ;CURSORSPALTE
C058 00           LENGTH .BYTE 0        ;LAENGE
C059 00           CODE   .BYTE 0        ;INPUT-CODE
C05A 00           COLOR  .BYTE 0        ;FARBE
C05B 00           ALLNUM .BYTE 0        ;ZAHL ERL. ZEICHEN
C05C 64           FIELDC .BYTE $64      ;FELD MARKIERER
C05D 00           OUTNUM .BYTE 0        ;ZAEHLER
C05E 00           VARADL .BYTE 0        ;ADR FUER VAR. LOW
C05F 00           VARADH .BYTE 0        ;ADR FUER VAR. HIGH
                  ALLSTR *=*+30         ;ERLAUBTE ZEICHEN
                  ;
                  ;----------------------------------------
                  ;      BEGINN NEUER INPUT-BEFEHL
                  ;
                  ;           CURSOR SETZEN
                  ;----------------------------------------
                  ;
C07E A2 00        ACCINP LDX #$00
C080 8E 5B C0            STX ALLNUM
C083 8E 59 C0            STX CODE
C086 20 9B B7            JSR GETBYT     ;ZEILE NACH X
C089 8A                  TXA
C08A 38                  SEC
C08B E9 01               SBC #$01
C08D C9 19               CMP #$19       ;> 25 ?
C08F B0 11               BCS CRSNO1     ;JA, ILLEGAL QUANT.
C091 8D 56 C0            STA CRSLIN     ;NEIN, SPEICHERN
C094 20 FD AE            JSR CKCOM      ;PRUEFT AUF KOMMA
C097 20 9E B7            JSR GETBY      ;SPALTE NACH X
C09A 8A                  TXA
C09B 38                  SEC
C09C E9 01               SBC #$01
C09E C9 28               CMP #$28       ;> 40 ?
C0A0 90 03               BCC CRSOK1     ;JA, WEITER
C0A2 4C 48 B2     CRSNO1 JMP ILQUAN     ;ILLEGAL QUANTITY
C0A5 A8           CRSOK1 TAY
C0A6 8C 57 C0            STY CRSCOL
C0A9 AE 56 C0            LDX CRSLIN
C0AC 20 0C E5            JSR CURSOR     ;CURSOR SETZEN
                  ;
                  ;----------------------------------------
                  ;          FELDLAENGE HOLEN
                  ;----------------------------------------
                  ;
C0AF 20 FD AE            JSR CKCOM      ;KOMMA ?
C0B2 20 9E B7            JSR GETBY      ;BYTE-WERT HOLEN
C0B5 E0 00               CPX #$00       ;0 ?
C0B7 F0 E9               BEQ CRSNO1     ;JA, ILLEGAL QUANT.
C0B9 8E 58 C0     LENGOK STX LENGTH     ;UND SPEICHERN
                  ;
                  ;----------------------------------------
                  ;   FARBE ODER TEXT HOLEN UND AUSGEBEN
                  ;----------------------------------------
                  ;
C0BC AC 86 02            LDY CURCOL     ;AKTUELLE FARBE
C0BF 8C 5A C0            STY COLOR      ;SPEICHERN
C0C2 20 FD AE            JSR CKCOM      ;KOMMA ?
C0C5 C9 39               CMP #$39       ;> "9" ?
C0C7 B0 11               BCS INTEXT     ;JA, DANN TEXT
C0C9 C9 2F               CMP #$2F       ;< "0" ?
C0CB 90 0D               BCC INTEXT     ;JA, DANN TEXT
C0CD 20 9E B7            JSR GETBY      ;FARBE HOLEN
C0D0 E0 10               CPX #$10       ;< 16 ?
C0D2 B0 CE               BCS CRSNO1     ;JA, ILLEGAL QUANT.
C0D4 8E 86 02            STX CURCOL     ;UND SPEICHERN
C0D7 20 FD AE            JSR CKCOM      ;KOMMA ?
C0DA C9 22        INTEXT CMP #$22       ;'"' ?
C0DC F0 03               BEQ OUTEXT     ;JA, DANN TEXT
C0DE 4C 00 C1            JMP CODEIN     ;NEIN, DANN CODE
C0E1 A0 01        OUTEXT LDY #$01       ;ZAEHLER = 1
C0E3 B1 7A        CONT3  LDA ($7A),Y    ;ZEICHEN HOLEN
C0E5 C9 22               CMP #$22       ;ENDE DES TEXTES ?
C0E7 F0 0A               BEQ GCODE      ;JA, DANN CODE
C0E9 20 D2 FF            JSR BSOUT      ;SONST AUSGEBEN
C0EC C8                  INY            ;ZAEHLER +1
C0ED 4C E3 C0            JMP CONT3      ;NAECHSTES ZEICHEN
C0F0 4C 99 AD     TYPERR JMP TYPMIS     ;TYPE-MISMATCH ERR
                  ;
                  ;----------------------------------------
                  ;       INPUT CODE HOLEN + PRUEFEN
                  ;----------------------------------------
                  ;
C0F3 38           GCODE  SEC            ;PROGRAMMZEIGER
C0F4 98                  TYA            ;UM ZAHL DER AUSGE-
C0F5 65 7A               ADC $7A        ;GEBENEN ZEICHEN
C0F7 90 02               BCC CONT2      ;ERHOEHEN
C0F9 E6 7B               INC $7B
C0FB 85 7A        CONT2  STA $7A
C0FD 20 FD AE            JSR CKCOM      ;KOMMA ?
C100 C9 5A        CODEIN CMP #'Z'       ;'Z' ?
C102 F0 0B               BEQ CODEOK     ;JA, OK
C104 C9 42               CMP #'B'       ;'B' ?
C106 F0 07               BEQ CODEOK     ;JA, OK
C108 C9 41               CMP #'A'       ;'A' ?
C10A F0 03               BEQ CODEOK     ;JA, OK
C10C 4C 08 AF            JMP SYNTAX     ;SONST SYNTAX ERROR
C10F 8D 59 C0     CODEOK STA CODE       ;UND SPEICHERN
```

Listing 2. Das ausführlich kommentierte Quell-Code-Listing des »Exinput 64«

```
                 ;
                 ;--------------------------------------
                 ;ERLAUBTE ZEICHEN HOLEN + ANZAHL PRUEFEN
                 ;--------------------------------------
                 ;
C112 20 73 00           JSR CHRGET      ;ZEICHEN HOLEN
C115 20 FD AE           JSR CKCOM       ;KOMMA ?
C118 A2 00              LDX #$00        ;ZAEHLER = 0
C11A C9 22              CMP #$22        ;" AM ANFANG ?
C11C D0 1E              BNE CREVAR      ;NEIN, WEITER
C11E 20 73 00    NEXT   JSR CHRGET      ;JA, ZEICHEN HOLEN
C121 C9 22              CMP #$22        ;" ZUM SCHLUSS
C123 F0 0E              BEQ ALLEND      ;JA, ENDE
C125 E0 1E              CPX #$1E        ;ZAEHLER = 30
C127 D0 03              BNE CONT23      ;NEIN, WEITER
C129 20 71 A5           JSR STRLON      ;SONST STR TOO LONG
C12C 9D 60 C0    CONT23 STA ALLSTR,X    ;ZEICHEN SPEICHERN
C12F E8                 INX             ;ZAEHLER + 1
C130 4C 1E C1           JMP NEXT        ;NAECHSTES ZEICHEN
C133 8E 5B C0    ALLEND STX ALLNUM      ;ZEICHENZAHL MERKEN
C136 20 73 00           JSR CHRGET      ;ZEICHEN HOLEN
C139 20 FD AE           JSR CKCOM       ;KOMMA ?
                 ;
                 ;--------------------------------------
                 ;       VARIABLE HOLEN
                 ;--------------------------------------
                 ;
C13C 20 8B B0    CREVAR JSR GETVAR      ;VARIABLE HOLEN
C13F A6 0D              LDX $0D         ;AUF STRING PRUEFEN
C141 F0 AD              BEQ TYPERR      ;SONST TYPE MISM.
C143 8D 5E C0           STA VARADL      ;ADR STRDESCRIPT.LO
C146 8C 5F C0           STY VARADH      ;ADRESSE HI
                 ;
                 ;--------------------------------------
                 ;     FELDMARKIERUNG AUSGEBEN
                 ;--------------------------------------
                 ;
C149 A0 00              LDY #$00        ;ZAEHLER = 0
C14B AD 5C C0    FIELD  LDA FIELDC      ;MARKIERUNGSZEICHEN
C14E C9 64              CMP #$64        ;HOLEN
C150 D0 02              BNE CONT28
C152 A9 A4              LDA #$A4
C154 20 D2 FF    CONT28 JSR BSOUT       ;UND AUSGEBEN
C157 C8                 INY             ;ZAEHLER ERHOEHEN
C158 CC 58 C0           CPY LENGTH      ;MIT LAENGE VERGL.
C15B D0 EE              BNE FIELD       ;KLEINER ?, WEITER
                 ;
                 ;--------------------------------------
                 ;  CURSOR AUF STARTPOSITION SETZEN
                 ;--------------------------------------
                 ;
C15D A0 00              LDY #$00
C15F A9 9D       CONT27 LDA #$9D        ;CRSR LINKS
C161 20 D2 FF           JSR BSOUT       ;AUSGEBEN
C164 C8                 INY
C165 CC 58 C0           CPY LENGTH      ;BIS LAENGE
C168 D0 F5              BNE CONT27
                 ;
                 ;--------------------------------------
                 ;     STARTPOSITION MERKEN
                 ;--------------------------------------
                 ;
C16A A4 D3              LDY $00D3       ;CURSOR SPALTE UND
C16C A6 D6              LDX $00D6       ;CURSOR ZEILE
C16E 8C 57 C0           STY CRSCOL      ;SPEICHERN
C171 8E 56 C0           STX CRSLIN
                 ;
                 ;--------------------------------------
                 ;   BILDSCHIRMPOSITION MERKEN
                 ;--------------------------------------
                 ;
C174 20 24 EA           JSR $EA24       ;ZEIGER FARB-RAM
C177 A5 D2              LDA $D2         ;AKTUELLE ZEILE HB
C179 85 FE              STA $FE         ;MERKEN
C17B AC 57 C0           LDY CRSCOL
C17E 98                 TYA
C17F 18                 CLC
C180 65 D1              ADC $D1         ;AKTUELLE ZEILE LB
C182 90 02              BCC CONT6       ;ADDIEREN
C184 E6 FE              INC $FE
C186 85 FD       CONT6  STA $FD         ;UND MERKEN
C188 A5 F4              LDA $F4         ;ZEIGER IN FARB-RAM
C18A 85 FC              STA $FC         ;HB MERKEN
C18C 98                 TYA
C18D 18                 CLC
C18E 65 F3              ADC $F3         ;ZEIGER LB ADDIEREN
C190 90 02              BCC CONT7
C192 E6 FC              INC $FC
C194 85 FB       CONT7  STA $FB         ;UND MERKEN
                 ;
                 ;--------------------------------------
                 ;     HAUPT EINGABESCHLEIFE
                 ;--------------------------------------
                 ;
C196 A9 00       RSTART LDA #$00
C198 8D 55 C0           STA COUNT       ;FELD ZAEHLER = 0
C19B 46 CC       INLOOP LSR $CC         ;CRSR FLAG EIN/AUS
C19D 46 CF              LSR $CF         ;CRSR EIN/AUS PHASE
C19F 20 E4 FF           JSR GETIN       ;ZEICHEN VON TASTAT
C1A2 F0 F7              BEQ INLOOP      ;NEIN, WARTEN
```

```
C1A4 48                 PHA             ;JA, ZEICHEN MERKEN
C1A5 E6 CC              INC $CC         ;CURSOR AUS
C1A7 A4 D3              LDY $D3         ;SPALTE HOLEN
C1A9 B1 D1              LDA ($D1),Y     ;ZEICHEN VON SCHIRM
C1AB 29 7F              AND #$7F        ;POSITIVIEREN UND
C1AD 91 D1              STA ($D1),Y     ;ZURUECK SPEICHERN
                 ;
                 ;--------------------------------------
                 ;   EINGEGEBENES ZEICHEN PRUEFEN
                 ;--------------------------------------
                 ;
C1AF A5 91              LDA $91         ;STOP FLAG
C1B1 C9 7F              CMP #$7F        ;CODE FUER STOP ?
C1B3 F0 41              BEQ NOTACC      ;NICHT ERLAUBT
C1B5 68                 PLA             ;ZEICHEN HOLEN
C1B6 C9 0D              CMP #$0D        ;RETURN
C1B8 D0 03              BNE CONT5       ;NEIN, WEITER
C1BA 4C B6 C3           JMP TAKEVA      ;JA, MIT SPACES
C1BD C9 8D       CONT5  CMP #$8D        ;SHIFT/RETURN
C1BF D0 03              BNE CONT8       ;NEIN, WEITER
C1C1 4C D0 C3           JMP MKSPAC      ;JA, OHNE SPACES
C1C4 C9 14       CONT8  CMP #$14        ;DELETE
C1C6 F0 37              BEQ MOVLFT      ;JA, WEITER
C1C8 C9 9D              CMP #$9D        ;CRSR LEFT
C1CA F0 33              BEQ MOVLFT      ;JA, WEITER
C1CC C9 1D              CMP #$1D        ;CRSR RIGHT
C1CE F0 38              BEQ PASS        ;JA,AUSGABE PRUEFEN
C1D0 C9 94              CMP #$94        ;INSERT
C1D2 F0 28              BEQ INSER       ;JA,AUSGABE PRUEFEN
C1D4 C9 91              CMP #$91        ;CRSR UP
C1D6 F0 2A              BEQ CRSUP       ;JA,AUSGABE PRUEFEN
C1D8 C9 13              CMP #$13        ;HOME
C1DA F0 29              BEQ HOM         ;JA, AUSGABE
C1DC C9 8C              CMP #$8C        ;F8
C1DE F0 2B              BEQ VARINF      ;JA, VARIABLE HOLEN
C1E0 C9 85              CMP #$85        ;F1-F7
C1E2 B0 03              BCS CONT25      ;NICHT
C1E4 4C EF C1           JMP CONT26      ;ANNEHMEN
C1E7 C9 8C       CONT25 CMP #$8C
C1E9 90 0B              BCC NOTACC
C1EB C9 93              CMP #$93        ;CLR/HOME
C1ED F0 16              BEQ HOM         ;JA, AUSGABE
C1EF C9 11       CONT26 CMP #$11        ;CRSR DOWN
C1F1 D0 1B              BNE CONT10      ;NEIN, WEITER
C1F3 4C 3A C3           JMP CRSRDN      ;JA, AUSGABE TESTEN
C1F6 20 A3 C4    NOTACC JSR SOUND       ;PIEPTON
C1F9 4C 9B C1           JMP INLOOP      ;ZUR EINGABE
C1FC 4C [illegible] C2    INSER  JMP INSERT
C1FF 4C 73 C2    MOVLFT JMP MOVLEF
C202 4C 14 C3    CRSUP  JMP CRSRUP
C205 4C 75 C3    HOM    JMP HOME
C208 4C 4E C2    PASS   JMP PASSED
C20B 4C C1 C4    VARINF JMP VARIN
                 ;
                 ;--------- INPUT-CODE PRUEFEN ----------
                 ;
C20E AE 59 C0    CONT10 LDX CODE        ;CODE HOLEN
C211 E0 41              CPX #$41        ;CODE = A
C213 F0 26              BEQ ACCALL      ;JA, OK
C215 C9 2E              CMP #$2E        ;ZEICHEN = .
C217 F0 22              BEQ ACCALL      ;JA, OK
C219 C9 2D              CMP #$2D        ;ZEICHEN = -
C21B F0 1E              BEQ ACCALL      ;JA, OK
C21D E0 5A              CPX #$5A        ;CODE = Z
C21F F0 0F              BEQ ACCNUM      ;JA, NUR ZIFFERN
C221 C9 20              CMP #$20        ;NEIN, BUCHSTABEN
C223 F0 16              BEQ ACCALL      ;LEERZEICHEN ?, OK
C225 C9 41              CMP #$41        ;ASCII < 65 ?
C227 90 CD              BCC NOTACC      ;JA, NICHT ERLAUBT
C229 C9 5B              CMP #$5B        ;ASCII < 91 ?
C22B 90 0E              BCC ACCALL      ;JA, OK
C22D 4C F6 C1           JMP NOTACC      ;NEIN,NICHT ERLAUBT
C230 C9 30       ACCNUM CMP #$30        ;ASCII  < 48 ?
C232 90 C2              BCC NOTACC      ;JA, NICHT ERLAUBT
C234 C9 3A              CMP #$3A        ;ASCII < 58 ?
C236 90 03              BCC ACCALL      ;JA, OK
C238 4C F6 C1           JMP NOTACC      ;NEIN,NICHT ERLAUBT
                 ;
                 ;------ ERLAUBTE ZEICHEN PRUEFEN -------
                 ;
C23B AE 5B C0    ACCALL LDX ALLNUM      ;ANZAHL HOLEN
C23E E0 00              CPX #$00        ;WENN 0
C240 F0 C6              BEQ PASS        ;DANN OK
C242 CA                 DEX             ;ZAEHLER -1
C243 DD 60 C0    CONT13 CMP ALLSTR,X    ;VERGLEICHEN
C246 F0 06              BEQ PASSED      ;GLEICH ? AUSGABE
C248 CA                 DEX             ;ZAEHLER -1
C249 10 F8              BPL CONT13      ;WENN NICHT DABEI
C24B 4C F6 C1           JMP NOTACC      ;NICHT ERLAUBT
                 ;
                 ;--------------------------------------
                 ;   ALLE TESTS OK, ZEICHEN AUSGEBEN
                 ;--------------------------------------
                 ;
C24E AE 55 C0    PASSED LDX COUNT       ;ZAEHLER HOLEN
C251 EC 58 C0           CPX LENGTH      ;MIT LAENGE VERGL.
C254 F0 0B              BEQ CONT14      ;GLEICH ?, WEITER
C256 EE 55 C0           INC COUNT       ;SONST ZAEHLER + 1
C259 AE 55 C0           LDX COUNT       ;NOCHMAL VERGL.
```

```
C25C EC 58 C0          CPX LENGTH     ;WENN NICHT GLEICH
C25F D0 09             BNE CAROUT     ;ZEICHEN AUSGEBEN
C261 C9 1D    CONT14   CMP #$1D       ;CRSR RECHTS ?
C263 F0 0B             BEQ CONT17     ;JA, NICHT MOEGLICH
C265 20 16 E7          JSR OUTSCR     ;NEIN, AUSGEBEN
C268 A9 9D             LDA #$9D       ;UND CURSOR EINE
C26A 20 16 E7 CAROUT   JSR OUTSCR     ;STELLE NACH RECHTS
C26D 4C 9B C1          JMP INLOOP     ;ZUR EINGABE
C270 4C F6 C1 CONT17   JMP NOTACC
              ;
              ;-----------------------------------------
              ;        CURSOR LINKS ODER DELETE
              ;-----------------------------------------
              ;
C273 AC 55 C0 MOVLEF   LDY COUNT      ;ZAEHLER HOLEN
C276 C0 00             CPY #$00       ;GLEICH 0 ?
C278 F0 F6             BEQ CONT17     ;NEIN,NICHT ERLAUBT
C27A CC 58 C0          CPY LENGTH     ;ZAEHLER = LAENGE
C27D D0 1F             BNE CONT19     ;NEIN, WEITER
C27F C9 9D             CMP #$9D       ;CRSR LINKS ?
C281 D0 0D             BNE DELET      ;NEIN, DANN DELETE
C283 AC 58 C0          LDY LENGTH     ;WENN LAENGE = 1
C286 C0 01             CPY #$01
C288 F0 38             BEQ LEN1       ;KEIN CURSOR LINKS
C28A CE 55 C0          DEC COUNT      ;SONST ZAEHLER -1
C28D 4C 9E C2          JMP CONT19     ;UND WEITER
C290 CE 55 C0 DELET    DEC COUNT      ;ZAEHLER -1
C293 AC 55 C0          LDY COUNT
C296 AD 5C C0          LDA FIELDC     ;MARKIERUNGSZEICHEN
C299 91 FD             STA ($FD),Y    ;ANST. DES LETZTEN
C29B 4C 9B C1          JMP INLOOP     ;ZEICHENS, ZURUECK
C29E CE 55 C0 CONT19   DEC COUNT      ;ZAEHLER -1
C2A1 C9 9D             CMP #$9D       ;CURSOR LINKS ?
C2A3 F0 C5             BEQ CAROUT     ;JA, DANN AUSGABE
C2A5 B1 FB    DELETE   LDA ($FB),Y    ;FARBE HOLEN
C2A7 AA                TAX
C2A8 B1 FD             LDA ($FD),Y    ;POSITION HOLEN
C2AA 88                DEY            ;EINE STELLE ZURECK
C2AB 91 FD             STA ($FD),Y    ;ZEICHEN AUSGEBEN
C2AD 8A                TXA
C2AE 91 FB             STA ($FB),Y    ;UND FARBE AUSGEBEN
C2B0 C8                INY            ;ZAEHLER ERHOEHEN
C2B1 C8                INY
C2B2 CC 58 C0          CPY LENGTH     ;LAENGE GANZ DURCH?
C2B5 90 EE             BCC DELETE     ;NEIN, NAECHSTES
C2B7 88                DEY            ;ZEICH. VERSCHIEBEN
C2B8 AD 5C C0          LDA FIELDC     ;JA, MARKIERUNG
C2BB 91 FD             STA ($FD),Y    ;ANS ENDE SETZEN
C2BD A9 9D    LEFOUT   LDA #$9D       ;CURSOR LINKS
C2BF 4C 6A C2          JMP CAROUT     ;AUSGEBEN, ZURUECK
C2C2 A0 00    LEN1     LDY #$00       ;ZAEHLER FUER DEL
C2C4 8C 55 C0          STY COUNT      ;ZURUECKSETZEN
C2C7 4C F6 C1          JMP NOTACC
              ;
              ;-----------------------------------------
              ;              INSERT
              ;-----------------------------------------
              ;
C2CA AC 58 C0 INSERT   LDY LENGTH     ;ZAEHLER = LAENGE
C2CD CE 58 C0          DEC LENGTH     ;LAENGE - 1
C2D0 88       CONT15   DEY            ;ZAEHLER -1
C2D1 C0 FF             CPY #$FF       ;ZAEHLER UEBER 0 ?
C2D3 F0 33             BEQ NOINST     ;JA, KEIN INSERT
C2D5 CC 55 C0          CPY COUNT      ;FELD ZAEHLER= INS-
C2D8 90 2E             BCC NOINST     ;ZAEHLER ?
C2DA B1 FD             LDA ($FD),Y    ;WENN GROESSER
C2DC CD 5C C0          CMP FIELDC     ;DANN INSERT
C2DF F0 EF             BEQ CONT15
C2E1 CC 58 C0          CPY LENGTH
C2E4 F0 22             BEQ NOINST
C2E6 B1 FB    NEXTIN   LDA ($FB),Y    ;FARBE HOLEN
C2E8 AA                TAX
C2E9 B1 FD             LDA ($FD),Y    ;ZEICHEN HOLEN
C2EB C8                INY            ;POSITION + 1
C2EC 91 FD             STA ($FD),Y    ;ZEICHEN ZURUECK
C2EE 8A                TXA
C2EF 91 FB             STA ($FB),Y    ;FARBE ZURUECK
C2F1 88                DEY
C2F2 88                DEY
C2F3 C0 FF             CPY #$FF       ;ALLE ZEICHEN ?
C2F5 F0 05             BEQ CONT16     ;JA, DANN WEITER
C2F7 CC 55 C0          CPY COUNT      ;NEIN, DANN
C2FA B0 EA             BCS NEXTIN     ;NAECHSTES ZEICHEN
C2FC C8       CONT16   INY
C2FD AD 5C C0          LDA FIELDC     ;MARKIERUNGSZEICHEN
C300 91 FD             STA ($FD),Y    ;ZWISCHEN 2 STRING-
C302 EE 58 C0 NOTINS   INC LENGTH     ;TEILE
C305 4C 9B C1          JMP INLOOP     ;ZUR EINGABE
C308 48       NOINST   PHA            ;KEIN INSERT
C309 8A                TXA
C30A 48                PHA
C30B 20 A3 C4          JSR SOUND      ;PIEPTON
C30E 68                PLA
C30F AA                TAX
C310 68                PLA
C311 4C 02 C3          JMP NOTINS     ;UND ZURUECK
              ;
              ;-----------------------------------------
              ;         CURSOR NACH OBEN
              ;-----------------------------------------
              ;
C314 AD 55 C0 CRSRUP   LDA COUNT      ;FELDZAEHLER
C317 38                SEC
C318 E9 28             SBC #$28       ;- 40
C31A 90 1B             BCC NOTUP      ;WENN < 0 NICHT
C31C AE 55 C0          LDX COUNT      ;MOEGLICH
C31F EC 58 C0          CPX LENGTH     ;ZAEHLER = LAENGE
C322 D0 0B             BNE CONT38     ;NEIN, WEITER
C324 8D 55 C0          STA COUNT      ;FELDZAEHLER MERKEN
C327 A9 1D             LDA #$1D       ;CURSOR RECHTS
C329 20 16 E7          JSR OUTSCR     ;AUSGEBEN
C32C 4C 32 C3          JMP CONT39
C32F 8D 55 C0 CONT38   STA COUNT      ;FELDZAEHLER MERKEN
C332 A9 91    CONT39   LDA #$91       ;CURSOR NACH OBEN
C334 4C 6A C2          JMP CAROUT     ;AUSGEBEN
C337 4C F6 C1 NOTUP    JMP NOTACC
              ;
              ;-----------------------------------------
              ;         CURSOR NACH UNTEN
              ;-----------------------------------------
              ;
C33A AD 58 C0 CRSRDN   LDA LENGTH     ;WENN LAENGE = 40
C33D C9 28             CMP #$28
C33F F0 0F             BEQ CONT36     ;NICHT MOEGLICH
C341 AD 55 C0          LDA COUNT
C344 18                CLC            ;WENN FELDZAEHLER
C345 69 28             ADC #$28       ;> 255, DANN NICHT
C347 B0 07             BCS CONT36     ;MOEGLICH
C349 CD 58 C0          CMP LENGTH     ;WENN FELDZAEHLER=
C34C F0 05             BEQ NOTOVE     ;LAENGE,1 POS LINKS
C34E 90 0E             BCC DOWNOK     ;SONST OK
C350 4C F6 C1 CONT36   JMP NOTACC
C353 8D 55 C0 NOTOVE   STA COUNT      ;FELDZAEHLER MERKEN
C356 A9 9D             LDA #$9D       ;CRSR LINKS
C358 20 16 E7          JSR OUTSCR     ;AUSGEBEN
C35B 4C 61 C3          JMP CONT21     ;WEITER
C35E 8D 55 C0 DOWNOK   STA COUNT
C361 A9 11    CONT21   LDA #$11       ;CURSOR NACH UNTEN
C363 20 16 E7          JSR OUTSCR     ;AUSGEBEN
C366 AE 58 C0          LDX LENGTH     ;WENN LETZTES FELD,
C369 CA                DEX            ;FELDZAEHLER
C36A EC 55 C0          CPX COUNT      ;KORRIGIEREN
C36D D0 03             BNE CONT37
C36F EE 55 C0          INC COUNT
C372 4C 9B C1 CONT37   JMP INLOOP     ;ZUR EINGABESCHL.
              ;
              ;-----------------------------------------
              ;              HOME
              ;-----------------------------------------
              ;
C375 48       HOME     PHA            ;ZEICHEN MERKEN
C376 AC 55 C0          LDY COUNT      ;FELDZAEHLER= 0
C379 F0 0E             BEQ CONT24     ;JA, DANN WEITER
C37B CC 58 C0          CPY LENGTH     ;WENN FELDZAEHLER
C37E D0 01             BNE CONT30     ;= LAENGE, DANN
C380 88                DEY            ;FELDZAEHLER -1
C381 A9 9D    CONT30   LDA #$9D       ;CURSOR LINKS BIS
C383 20 D2 FF          JSR BSOUT      ;ZUR ANFANGSPOSIT.
C386 88                DEY            ;AUSGEBEN
C387 D0 F8             BNE CONT30     ;WENN >0 WEITER
C389 68       CONT24   PLA            ;ZEICHEN HOLEN
C38A C9 93             CMP #$93       ;CLR/HOME ?
C38C F0 0A             BEQ CLRHOM     ;JA, WEITER
C38E C9 FF             CMP #$FF       ;HOME NACH ALTE
C390 F0 03             BEQ VARIIN     ;VARIABLE INS FELD
C392 4C 96 C1          JMP RSTART     ;ZUR EINGABESCHL.
C395 4C C6 C4 VARIIN   JMP CONT11
              ;
              ;-----------------------------------------
              ;            CLR/HOME
              ;-----------------------------------------
              ;
C398 AC 58 C0 CLRHOM   LDY LENGTH     ;ZAEHLER = LAENGE
C39B AD 5C C0 CONT32   LDA FIELDC     ;ALLE ZEICHEN DURCH
C39E C9 64             CMP #$64       ;MARKIERUNGSZEICHEN
C3A0 D0 02             BNE CONT31
C3A2 A9 A4             LDA #$A4
C3A4 20 D2 FF CONT31   JSR BSOUT      ;ERSETZEN
C3A7 88                DEY
C3A8 D0 F1             BNE CONT32     ;UND CURSOR
C3AA A9 13             LDA #$13
C3AC 48                PHA
C3AD AC 58 C0          LDY LENGTH     ;NACH HOME
C3B0 4C 81 C3          JMP CONT30     ;BRINGEN
C3B3 4C 96 C1 RESTAR   JMP RSTART     ;ZUR EINGABESCHL.
              ;
              ;-----------------------------------------
              ;   AUSLESE DER MARKIERUNGSZEICHEN
              ;-----------------------------------------
              ;
C3B6 AC 58 C0 TAKEVA   LDY LENGTH     ;ZAEHLER= LAENGE
C3B9 4C C0 C3          JMP BCKPRT     ;MARKIERUNGSZEICHEN
C3BC A9 20    ELIMIN   LDA #$20       ;AM ENDE DURCH LEER
C3BE 91 FD             STA ($FD),Y    ;-ZEICHEN ERSETZEN
C3C0 88       BCKPRT   DEY
C3C1 C0 FF             CPY #$FF
C3C3 F0 07             BEQ CONT34     ;LAENGE ERREICHT,
C3C5 B1 FD             LDA ($FD),Y
```

```
C3C7 CD 5C C0          CMP FIELDC
C3CA F0 F0             BEQ ELIMIN
C3CC C8         CONT34 INY
C3CD 8C 58 C0          STY LENGTH          ;NEUE LAENGE MERKEN
C3D0 AC 58 C0   MKSPAC LDY LENGTH          ;UND MARKIERUNG
C3D3 88         MAKSPC DEY                 ;ZWISCHEN ANDEREN
C3D4 C0 FF             CPY #$FF            ;ZEICHEN ENTFERNEN
C3D6 F0 0E             BEQ TAKVAR          ;FERTIG,ZU VARIABLE
C3D8 B1 FD             LDA ($FD),Y         ;ZEICHEN HOLEN
C3DA CD 5C C0          CMP FIELDC          ;MARKIERUNG ?
C3DD D0 F4             BNE MAKSPC
C3DF A9 20             LDA #$20
C3E1 91 FD             STA ($FD),Y         ;JA, ERSETZEN
C3E3 4C D3 C3          JMP MAKSPC          ;UND WEITER

                ;
                ;-----------------------------------------
                ;  VARIABLE ERSTELLEN + STRING KOPIEREN
                ;-----------------------------------------
                ;
C3E6 AE 5A C0   TAKVAR LDX COLOR           ;FARBE ZURUECK
C3E9 8E 86 02          STX CURCOL
C3EC AE 58 C0          LDX LENGTH          ;WENN NEUE LAENGE 0
C3EF F0 4B             BEQ CONT29          ;KEINE VARIABLE
C3F1 20 A3 B6          JSR FRESTR          ;AUSWERTUNG DES
C3F4 AD 5F C0          LDA VARADH          ;STRINGAUSDRUCKS
C3F7 85 65             STA $65             ;ZEIGER AUF
C3F9 AD 5E C0          LDA VARADL          ;STRINGDESCRIPTOR
C3FC 85 64             STA $64
C3FE AD 58 C0          LDA LENGTH
C401 20 75 B4          JSR STRPOI          ;BERECHNET STRZEIG.
C404 84 FB             STY $FB             ;STRZEIGER KOPIEREN
C406 A0 00             LDY #$00
C408 91 64             STA ($64),Y         ;LAENGE
C40A C8                INY
C40B 8A                TXA
C40C 91 64             STA ($64),Y         ;ADRESSE LOW
C40E C8                INY
C40F A5 FB             LDA $FB
C411 91 64             STA ($64),Y         ;ADRESSE HIGH
C413 A0 01             LDY #$01
C415 B1 64             LDA ($64),Y
C417 48                PHA
C418 C8                INY
C419 B1 64             LDA ($64),Y         ;ADR DES STRING
C41B 85 65             STA $65             ;NACH $64/$65
C41D 68                PLA
C41E 85 64             STA $64

                ;
                ;-----------------------------------------
                ;     STING IN VARIABLE EINLESEN
                ;-----------------------------------------
                ;
C420 AC 58 C0          LDY LENGTH          ;ZAEHLER = LAENGE
C423 88         READIN DEY
C424 B1 FD             LDA ($FD),Y         ;ZEICHEN LESEN
C426 29 7F             AND #$7F            ;BILDSCHIRMCODE
C428 C9 20             CMP #$20            ;NACH ASCII
C42A 90 19             BCC CONT40          ;WANDELN
C42C C9 40             CMP #$40
C42E 90 06             BCC STOVAR
C430 C9 60             CMP #$60
C432 90 15             BCC CONT41
C434 69 3F             ADC #$3F            ;UND NACH ADR AUS
C436 91 64      STOVAR STA ($64),Y         ;$64/$65 SCHREIBEN
C438 C0 00             CPY #$00            ;ALLES GELESEN
C43A D0 E7             BNE READIN          ;NEIN, WEITER
C43C AE 56 C0   CONT29 LDX CRSLIN          ;JA, CURSOR SETZEN
C43F 20 0C E5          JSR CURSOR          ;UND ZURUECK ZUR
C442 4C AE A7          JMP INTERP          ;INTERPRETERSCHL.
                ;-----------------------------------------
C445 09 40      CONT40 ORA #$40            ;BILDSCHIRM
C447 D0 ED             BNE STOVAR          ;NACH
C449 09 20      CONT41 ORA #$20            ;ASCII
C44B D0 E9             BNE STOVAR
                ;
                ;*****************************************
                ;-----------------------------------------
                ;    AUF NEUEN PRINT-BEFEHL PRUEFEN
                ;-----------------------------------------
                ;
C44D 20 73 00   PRINT  JSR CHRGET          ;ZEICHEN HOLEN
C450 C9 B1             CMP #$B1            ;> ?
C452 F0 06             BEQ ACCPRI          ;JA, NEUER BEFEHL
C454 20 A0 AA          JSR BPRINT          ;SONST BASIC-PRINT
C457 4C AE A7          JMP INTERP          ;INTERPRETERSCHL.
                ;
                ;-----------------------------------------
                ;         NEUER PRINT-BEFEHL
                ;-----------------------------------------
                ;
C45A 20 9B B7   ACCPRI JSR GETBYT          ;CURSORPOSITION
C45D 8A                TXA                 ;HOLEN, SIEHE
C45E 38                SEC                 ;INPUT-BEFEHL
C45F E9 01             SBC #$01
C461 C9 19             CMP #$19            ;> 25
C463 B0 0F             BCS CRSNO2
C465 48                PHA
C466 20 FD AE          JSR CKCOM
C469 20 9E B7          JSR GETBY
C46C 8A                TXA
C46D 38                SEC
C46E E9 01             SBC #$01
C470 C9 28             CMP #$28            ;> 40
C472 90 03             BCC CRSOK2
C474 4C 48 B2   CRSNO2 JMP ILQUAN
C477 A8         CRSOK2 TAY
C478 68                PLA
C479 AA                TAX
C47A 20 0C E5          JSR CURSOR          ;CURSOR SETZEN
                ;
                ;-----------------------------------------
                ;  FARBE HOLEN ODER TEXT AUSGEBEN
                ;-----------------------------------------
                ;
C47D AD 86 02          LDA CURCOL          ;AKTUELLE FARBE
C480 8D 5A C0          STA COLOR           ;MERKEN
C483 20 FD AE          JSR CKCOM           ;KOMMA ?
C486 C9 39             CMP #$39            ;ASCII < 57 ?
C488 B0 0D             BCS PRICOM          ;NEIN, DANN PRINT
C48A C9 2F             CMP #$2F            ;ASCII > 47 ?
C48C 90 09             BCC PRICOM          ;NEIN, DANN PRINT
C48E 20 9E B7          JSR GETBY           ;JA, BYTE-WERT
C491 8E 86 02          STX CURCOL          ;HOLEN + SPEICHERN
C494 20 FD AE          JSR CKCOM           ;KOMMA ?
C497 20 A0 AA   PRICOM JSR BPRINT          ;BASIC PRINT-BEFEHL
C49A AD 5A C0          LDA COLOR           ;FARBE WIEDER
C49D 8D 86 02          STA CURCOL          ;ZURUECKSTELLEN UND
C4A0 4C AE A7          JMP INTERP          ;INTERPRETERSCHL.
                ;
                ;-----------------------------------------
                ;              PIEPTON
                ;-----------------------------------------
                ;
C4A3 A2 41      SOUND  LDX #$41            ;WELLENFORM 65
C4A5 8E 04 D4          STX $D404           ;SETZEN
C4A8 A2 90             LDX #$90            ;ZAEHLER SETZEN
C4AA 86 02             STX $02             ;UND HERUNTER
C4AC CA         LOOP   DEX                 ;ZAEHLEN
C4AD D0 FD             BNE LOOP
C4AF C6 02             DEC $02
C4B1 F0 05             BEQ OFF
C4B3 A2 FF             LDX #$FF
C4B5 4C AC C4          JMP LOOP
C4B8 A2 00      OFF    LDX #$00            ;WELLENF. LOESCHEN
C4BA 8E 04 D4          STX $D404
C4BD 60                RTS                 ;ZURUECK
C4BE 4C 96 C1   NOVAR  JMP RSTART
                ;
                ;-----------------------------------------
                ;     ALTE VARIABLE INS FELD GEBEN
                ;-----------------------------------------
                ;
C4C1 A9 FF      VARIN  LDA #$FF            ;CURSOR NACH HOME
C4C3 4C 75 C3          JMP HOME
C4C6 A0 00      CONT11 LDY #$00            ;ZAEHLER = 0
C4C8 B1 47      CONT12 LDA ($47),Y         ;STRING-DESCRIPTOR
C4CA 99 61 00          STA $61,Y           ;NACH $61-$63 KOP.
C4CD C8                INY                 ;ZAEHLER +1
C4CE C0 03             CPY #$03
C4D0 D0 F6             BNE CONT12
C4D2 A5 61             LDA $61             ;LAENGE DES ALTEN
C4D4 CD 58 C0          CMP LENGTH          ;STR MIT DER DES
C4D7 90 03             BCC CONT9           ;FELDES VERGLEICHEN
C4D9 AD 58 C0          LDA LENGTH          ;ZU LANG,DANN FELD-
C4DC 8D 5D C0   CONT9  STA OUTNUM          ;LAENGE ALS ZAEHLER
C4DF A0 00             LDY #$00
C4E1 C9 00             CMP #$00            ;FELD-LAENGE=0 ?
C4E3 F0 D9             BEQ NOVAR           ;NICHTS INS FELD
C4E5 B1 62      STROUT LDA ($62),Y         ;VARIABLE LESEN UND
C4E7 C9 20             CMP #$20            ;LEERZEICHEN DURCH
C4E9 D0 07             BNE CONT18
C4EB AD 5C C0          LDA FIELDC          ;MARKIERUNGSZEICHEN
C4EE 91 FD             STA ($FD),Y         ;ERSETZEN
C4F0 A9 1D             LDA #$1D            ;CURSOR RECHTS
C4F2 20 D2 FF   CONT18 JSR BSOUT           ;AUSGEBEN
C4F5 C8                INY
C4F6 18                CLC
C4F7 CC 5D C0          CPY OUTNUM          ;GANZE VARIABLE ?
C4FA B0 03             BCS FILSPC          ;JA,REST AUFFUELLEN
C4FC 4C E5 C4          JMP STROUT          ;NEIN, WEITER
C4FF AD 5C C0   FILSPC LDA FIELDC          ;RESTLICHE LAENGE
C502 CC 58 C0   CONT4  CPY LENGTH          ;MIT MARKIERUNGS-
C505 B0 06             BCS ALLOUT          ;ZEICHEN
C507 91 FD             STA ($FD),Y         ;AUFFUELLEN
C509 C8                INY
C50A 4C 02 C5          JMP CONT4
C50D AC 5D C0   ALLOUT LDY OUTNUM          ;UND CURSOR
C510 A9 9D      CONT35 LDA #$9D            ;WIEDER
C512 20 D2 FF          JSR BSOUT           ;NACH HOME
C515 88                DEY                 ;BRINGEN
C516 C0 00             CPY #$00
C518 D0 F6             BNE CONT35
C51A 4C 96 C1          JMP RSTART          ;ZUR EINGABESCHL.
```

Listing 2. Das ausführlich kommentierte Quell-Code-Listing des »Exinput 64« (Schluß)

Directory auf Knopfdruck

Häufig steht man vor dem Problem, Programme direkt, also ohne SYS-Aufrufe, über einen Tastencode zu aktivieren. Wir zeigen Ihnen, wie man ein Directory auf dem Bildschirm ausgibt, ohne ein vorhandenes Basic-Programm zu zerstören.

Das Programm »Directory auf Knopfdruck« (siehe Listing 1) kann, nachdem es mit Hypra-Ass assembliert wurde, mit dem Befehl SYS 49152 aktiviert werden. Den erzeugten Maschinencode zeigt Listing 2, das auch direkt mit dem MSE eingegeben werden kann. Solange kein Reset ausgelöst oder die RUN/STOP-Taste gedrückt wird, erscheint bei der Tastenkombination »CTRL-D« das Directory auf dem Bildschirm, ohne ein eventuell im RAM stehendes Basic-Programm zu zerstören.

Der hier vorgestellte Lösungsweg, ein beliebiges Programm auf Tastendruck aufzurufen, hat den Vorteil, daß er allgemeingültig ist und nicht nur auf die Ausgabe des Directorys beschränkt ist.

Im allgemeinen müßte das Betriebssystem geändert werden, um eine zusätzliche Tastaturabfrage einzubinden. Aber wir haben Glück, denn einige Routinen im Betriebssystem werden über einen sogenannten Sprungvektor aufgerufen. Alle zugänglichen Sprungvektoren, die sich ändern lassen, befinden sich in der dritten Page, also im Bereich von $311 bis $333, jeweils im Low-/High-Byte-Format. Der zum Einbinden einer Tastaturabfrage geeignetste Vektor ist der Interruptvektor, denn diese sogenannte Interrupt-Routine wird 60mal pro Sekunde angesprungen. Ein Sprungvektor, der in diese Routine verzweigt, befindet sich in den Speicherzellen $314/$315. Die Speicherzellen enthalten im Normalzustand den Wert $31/$EA. Das bedeutet, daß immer, wenn ein Interrupt ausgelöst wird, zu einer Routine verzweigt wird, die bei $EA31 liegt. Schreibt man nun in die Speicherzellen $314/$315 einen anderen Wert, wird bei einem Interrupt nicht mehr zur Adresse $EA31, sondern zu der Adresse verzweigt, die in den Speicherzellen $314/315 steht, wobei die Adresse $314 das Low- und die Speicherzelle $315 das High-Byte der anzuspringenden, neuen Interrupt-Routine enthält. Damit sich der Vektor aber verändern läßt, muß zuerst der normale Interrupt gesperrt werden, denn sonst könnte folgendes passieren: Angenommen, es wird ein Interrupt ausgelöst, nachdem erst das Low-Byte in die Adresse $314 geschrieben wurde, dann nimmt der Computer an, daß die neue Interrupt-Routine bei $EA00 plus dem in Adresse $314 stehendem Low-Byte liegt, verzweigt an diese Adresse und stürzt ab.

Mit dem Maschinenbefehl »SEI« für SEt Interruptflag läßt sich nun der prozessorinterne Interrupt ausschalten. Dies wird in der Zeile 230 im Listing 1 realisiert. Die Zeilen 240 bis 270 sorgen dafür, daß die Startadresse der neuen Interrupt-Routine in die entsprechenden Adressen kommt. Eine Bemerkung noch zu den Zeilen 240 und 270. Durch das »kleiner«- beziehungsweise »größer«-Zeichen läßt sich eine 16-Bit-Adresse (in diesem Fall »INTER«) in zwei 8-Bit-Zahlen splitten. Durch das »kleiner«-Zeichen erhält man das Low- und entsprechend für das »größer«-Zeichen das High-Byte.

Der folgende »CLI«-Befehl (CLear Interruptflag) läßt nun den normalen Interrupt wieder zu. Nur wird jetzt, wenn ein Interrupt stattfindet, nicht mehr zur Adresse $EA31, sondern zur Routine »INTER« verzweigt. Die Aufgabe des Teilprogramms »INTER« ist es, die »CTRL«- und »D«-Taste abzufragen. Sind beide Tasten gleichzeitig gedrückt, wird nach »DIR« verzweigt, ansonsten in der Zeile 420 zum normalen Interrupt nach $EA31. Die Zeilen 370 und 380 beziehungsweise 400 und 410 sperren auch den Interrupt, aber auf eine etwas andere Art.

Sie werden sich vielleicht fragen, warum der Interrupt überhaupt gesperrt werden muß. Die Antwort auf diese Frage ist leicht zu geben. Die Bearbeitung der Routine »DIR« dauert länger als 1/60 Sekunde. Dadurch würde bereits ein neuer Interrupt ausgelöst, bevor der alte abgearbeitet wäre. Die Art, wie der Interrupt abgeschaltet wird, wurde mit Absicht so gewählt, da einige Routinen im Betriebssystem den Interrupt durch den CLI-Befehl wieder zulassen, so auch die Routine »CHROUT«, die ein Zeichen auf einem Gerät, hier dem Bildschirm, ausgibt. Durch die Befehle »LDA #TIMERAUS« und »STA ICR« wird eigentlich nicht der Interrupt abgeschaltet, sondern der Timer gestoppt, der den Interrupt auslöst. Doch nun zum eigentlichen »DIR«-Programm.

Die Zeilen 450 bis 550 entsprechen dem Basic-Befehl »OPEN 1,8,0,"$"«. Dabei liegt das Augenmerk auf der Sekundäradresse »0« und dem Dollarzeichen. Das Dollarzeichen als Filename bewirkt, daß das Directory eingelesen wird und zwar einschließlich Track 18 Sektor 0. Durch die »0« als Sekundäradresse wird das Directory schon soweit aufbereitet, daß nur noch die Zeichen, die sich nach einem »LOAD"$",8« tatsächlich im RAM befinden, zum Computer gesendet werden, inklusive Filetyp, Steuerzeichen, Leerzeichen und Anzahl der Blöcke pro File. Leider enthält ein vom Floppy-Laufwerk gesendeter Directory-Eintrag auch noch Platzhalter für die Zeilenlinkadressen, die wir natürlich nicht brauchen und folglich ausblenden beziehungsweise überlesen müssen. Eine vom Floppy-Laufwerk gesendete Directory-Zeile hat folgendes Format:

LI LI AB SP "filename SP" SP FT

LI = Linkbyte; AB = Anzahl der Blöcke pro File im Low/High-Byte-Format; SP = Leerzeichen; FT = Filetyp im Klartext.

Das einzige, was zu tun bleibt, ist die Ausgabe »RETURN« am Ende jeder Directory-Zeile (Zeile 811/812). Außerdem muß die Information, die die Anzahl der Blöcke pro File enthält, in dezimale Schreibweise umgerechnet und auf dem Bildschirm ausgegeben werden. Das ist aber kein Problem, denn das Betriebssystem stellt eine solche Routine »INTOUT=$BDCD« zur Verfügung. Wenn sie aufgerufen wird, muß der Akku das High- und das X-Register das Low-Byte der auszugebenden Zahl enthalten. Das Low-Byte wird in Zeile 590 an das X-Register übergeben. Das High-Byte dagegen wird in Zeile 630 durch das Unterprogramm »CHRIN« eingelesen und steht automatisch im Akku.

Zum Programmablauf

Die Zeilen 570 bis 620 überlesen 4 Byte beim ersten Durchlauf. Bei allen weiteren Durchläufen werden immer nur die ersten 2 Byte überlesen, die die Zeilenlinkadresse enthalten. Beim ersten Durchlauf wird neben der Zeilenlinkadresse auch die Startadresse ($0801) gesendet, die ebenfalls ausgeblendet werden muß, darum 4 Byte. Bei einem Sprung auf das Label »ROWOUT« enthält das Y-Register die Anzahl der Byte, die überlesen werden sollen plus 1. Wird die Schleife in Zeile 620 verlassen, enthält das X-Register das Low-Byte der Zahl, die die Anzahl der Blöcke pro File kennzeichnet. Das High-Byte wird in Zeile 630 eingelesen und an den Akku übergeben. Die nächste Schleife, die bei dem Label »NEXTCHR« beginnt, gibt den Rest eines Directory-Eintrags auf dem Bildschirm aus. Der »CMP«-Befehl in Zeile 730 überprüft, ob eine »0«, die das Ende des Eintrags kennzeichnet, gesendet wurde oder nicht. Bei einer »0« wird nach »LASTCHR« verzweigt, ein »RETURN« ausgegeben und zum Label »ROWOUT« gesprungen. (ah)

```
10     -.EQ IRQVEKLO = $314
20     -.EQ IRQVEKHI = $315
30     -.EQ SHIFTFLAG = $28D
40     -.EQ TIMERAUS = $80
50     -.EQ TIMEREIN = 129
60     -.EQ ICR = $DC0E
70     -.EQ INTERRUPTALT = $EA31
80     -.EQ SETNAM = $FFBD
90     -.EQ SETLFS = $FFBA
100    -.EQ OPEN = $FFC0
110    -.EQ CHKIN = $FFC6
120    -.EQ CHRIN = $FFCF
130    -.EQ INTOUT = $BDCD
140    -.EQ CHROUT = $FFD2
150    -.EQ CLOSE = $FFC3
151    -.EQ CLRCHN = $FFCC
160    -.EQ STATUS = $90
170    -.EQ RETTEN = $FB
180    -.EQ TASTENCODE = $C5
190    -;
200    -;
210    -.BA $C000
220    -;
221    -;********************************
222    -;* INTERRUPTVEKTOR NEU SETZEN    *
223    -;********************************
224    -;
230    -          SEI
240    -          LDA #<(INTER)
250    -          STA IRQVEKLO
260    -          LDA #>(INTER)
270    -          STA IRQVEKHI
280    -          CLI
290    -          RTS
300    -;
301    -;********************************
302    -;*     NEUE INTERRUPTROUTINE     *
303    -;********************************
304    -;
310    -INTER     LDA TASTENCODE
320    -          CMP #18          ;IST DIE TASTE >>D<< GEDRUECKT?
330    -          BNE INTERE       ;WENN NEIN DANN ZUM NORMALEN INTERRUPT
340    -          LDA SHIFTFLAG
350    -          CMP #4           ;IST >>CTRL<< GEDRUECKT?
360    -          BNE INTERE       ;WENN NEIN DANN ZUM NORMALEN INTERRUPT
370    -          LDA #TIMERAUS    ;TIMER AUSSCHALTEN/INTERRUPT SPERREN
380    -          STA ICR
390    -          JSR DIR          ;UND ZUM UNTERPROGRAMM DIR
400    -          LDA #TIMEREIN    ;TIMER EINSCHALTEN/INTERRUPT ZULASSEN
410    -          STA ICR
420    -INTERE    JMP INTERRUPTALT
430    -;
440    -;
441    -;********************************
442    -;*          DIRECTORY            *
443    -;*          AUSGEBEN             *
444    -;********************************
445    -;
450    -DIR       LDA #1           ;LAENGE DES FILENAMENS
460    -          LDY #>(NAME)
470    -          LDX #<(NAME)
480    -          JSR SETNAM       ;FILENAMENPARAMETERSETZEN
490    -          LDA #1           ;LOGISCHE FILENUMMER = 1
500    -          LDX #8           ;GERAETEADRESSE = 8
510    -          LDY #0           ;SEKUNDAERADRESSE = 0
520    -          JSR SETLFS       ;LOGISCHE FILENUMMER UND SEKUNDAERADR. SETZEN
530    -          JSR OPEN
540    -          LDX #1           ;LOGISCHE FILENUMMER
550    -          JSR CHKIN        ;EINGABEGERAET SETZEN
560    -          LDY #5           ;DIE ERSTEN 5 ZEICHEN UEBERLESEN
570    -ROWOUT    STY RETTEN
580    -          JSR CHRIN        ;EINGABE EINES ZEICHENS
590    -          TAX              ;LO-BYTE BLOCKANZAHL INS X-REGISTER
600    -          LDY RETTEN
610    -          DEY
620    -          BNE ROWOUT
630    -          JSR CHRIN        ;HI-BYTE BLOCKANZAHL IN DEN AKKU
640    -          LDY STATUS       ;IST FILEENDE SCHON ERREICHT?
650    -          BNE ENDE         ;WENN JA DANN FERTIG
670    -          JSR INTOUT       ;ANZAHL DER BLOECKE AUSGEBEN (X/AKKU)
690    -          LDA #" "         ;BLANK AUSGEBEN
700    -          JSR CHROUT
710    -NEXTCHR   JSR CHRIN        ;NAECHSTES ZEICHEN HOLEN
720    -          STA RETTEN       ;UND RETTEN
730    -          CMP #0           ;IST FILEEINTRAG AUSGEGEBEN?
740    -          BEQ LASTCHR      ;WENN JA DANN RETURN AUSGEBEN
790    -          LDA RETTEN       ;ANSONSTEN MUSS NOCH DAS LETZTE ZEICHEN
800    -          JSR CHROUT       ;AUSGEGEBEN WERDEN
810    -          JMP NEXTCHR
811    -LASTCHR   LDA #13          ;>>RETURN<< AUSGEBEN
812    -          JSR CHROUT
813    -          LDY #$3          ;3 ZEICHEN UEBERLESEN
814    -          JMP ROWOUT
820    -ENDE      JSR CLRCHN       ;EIN- UND AUSGABEKANAELE
830    -          LDA #1           ;UND LOGISCHES FILE SCHLIESSEN
840    -          JSR CLOSE
850    -          RTS
860    -;
870    -NAME      .TX "$"
```

Listing 1. »Directory auf Knopfdruck«. Der Quelltext wurde mit Hypra-Ass erstellt.

```
programm : direktory.obj    c000 c085
--------------------------------------

c000 : 78 a9 0d 8d 14 03 a9 c0  c3
c008 : 8d 15 03 58 60 a5 c5 c9  ca
c010 : 12 d0 14 ad 8d 02 c9 04  5d
c018 : d0 0d a9 80 8d 0e dc 20  e6
c020 : 2a c0 a9 81 8d 0e dc 4c  9a
c028 : 31 ea a9 01 a0 c0 a2 84  fc
c030 : 20 bd ff a9 01 a2 08 a0  ea
c038 : 00 20 ba ff 20 c0 ff a2  44
c040 : 01 20 c6 ff a0 05 84 fb  3f
c048 : 20 cf ff aa a4 fb 88 d0  93
c050 : f5 20 cf ff a4 90 d0 23  a1
c058 : 20 cd bd a9 20 20 d2 ff  52
c060 : 20 cf ff 85 fb c9 00 f0  08
c068 : 08 a5 fb 20 d2 ff 4c 60  65
c070 : c0 a9 0d 20 d2 ff a0 03  02
c078 : 4c 46 c0 20 cc ff a9 01  91
c080 : 20 c3 ff 60 24 ff ff ff  d0
```

Listing 2. »Directory auf Knopfdruck« muß mit dem MSE eingegeben werden

Tips und Tricks zu Hypra-Ass

Hypra-Ass, ein Assembler der Spitzenklasse beherrscht alles, was zum Programmieren in Maschinensprache erforderlich ist. Hier wollen wir Ihnen anhand vieler Beispiele zeigen, was in ihm steckt und was er wirklich leistet.

Hypra-Ass ist einer der leistungsfähigsten Assembler die zur Zeit auf dem Markt sind. Eine seiner hervorstechendsten Eigenschaften ist der integrierte Editor mit einer formatierenden LIST-Routine. Aber gerade durch diese Eigenschaft weicht die Bedienung vom normalen Basic-Editor ab. Dadurch traten bei vielen Lesern Schwierigkeiten auf, die hier im einzelnen behandelt werden.

Im Gegensatz zum Basic-Editor kann unter Hypra-Ass eine Zeile nicht dadurch gelöscht werden, daß nur die Zeilennummer eingegeben und anschließend RETURN gedrückt wird. Bei Hypra-Ass ist unbedingt darauf zu achten, daß hinter der Zeilennummer ein Minuszeichen eingegeben wird. Drückt man nun die RETURN-Taste ist die Zeile auch verschwunden. Da aber dieses Minuszeichen hinter der Zeilennummer meistens vergessen wird, ist es empfehlenswert, nicht nur Zeilenbereiche, sondern auch einzelne Zeilen mit dem Editor-Befehl »/D zeilennummer« zu löschen, bis auf die Zeile »0«, die sich mit dem »/D«-Befehl nicht löschen läßt. In diesem Fall geben Sie bitte »0-« <RETURN> ein.

Ein kleiner Fehler tritt beim Sortieren der Symboltabelle auf. Hypra-Ass stürzt ab, wenn die Symboltabelle genau 36, 73, 109 (und so weiter) Variablen oder Label enthält. Der Fehler liegt in den Speicherzellen $1EB8 bis $1EBB. Hier wurden zwei Branch-Befehle vertauscht. Es muß richtig lauten:

```
1EB8 90 D0    BCC 1EA8
1EBA D0 04    BNE 1EC0
```

Diese Änderung kann unmittelbar mit einem Monitor in die entsprechenden Speicherzellen geschrieben und anschließend gespeichert werden. Sollten Sie keinen Monitor haben, dann geben Sie bitte den folgenden Quelltext ein:

```
 10 -.BA $C000
;STARTADRESSE = $C000
 20 -          LDY #0
 30 - LBL      LDA TAB,Y
;KORREKTUREN VORNEHMEN
 40 -          STA $1EB8,Y
 50 -          INY
 60 -          CPY #4
 70 -          BNE LBL
 80 -.EQ SOURCESTART = $1FD8 ;UND DIE KORRIGIERTE
 90 -.EQ NAMLEN = 12         ;VERSION SPEICHERN
100-           LDA #1
110-           LDX #8
120-           STA $FE
130-           STX $FF
140-           LDA #8
150-           JSR $FFBA
160-           LDA #NAMLEN
170-           LDX #<(NAME)
180-           LDY #>(NAME)
190-           JSR $FFBD
200-           LDA # $FE
210-           LDX #<(SOURCESTART)
220-           LDY #>(SOURCESTART)
230-           JMP $FFD8
240-;
250-NAME       .TX "HYPRA-ASS.V1 "
260-TAB        .BY $90,$D0,$D0,$04
```

Nach dem Assemblieren wird mit SYS 49152 <RETURN> Hypra-Ass geändert und unter dem neuen Namen »Hypra-Ass.V1« auf Diskette gespeichert.

Der »/A«-Befehl zur automatischen Zeilennumerierung reagiert auch recht sensibel. Wird mit diesem Befehl gearbeitet, darf der Cursor mit den entsprechenden Steuertasten auf keinen Fall auf eine andere Zeile gesetzt und RETURN gedrückt werden. Sollte das versehentlich doch einmal passieren, läßt sich die so entstandene, ewas seltsam aussehende Zeile mit dem »/D«-Befehl problemlos löschen.

Diskettenbefehle können mit dem Editorbefehl »/@« zum Floppy-Laufwerk gesendet werden. Hinter den Editorbefehl werden dann die Diskettenbefehle unmittelbar angehängt. So formatiert der Befehl »/@N:NEWDISK,ND« eine neue Diskette.

Der Editor

Eine feine Sache ist auch das Arbeiten mit dem »/P«-Befehl, der dazu dient, Arbeitsseiten beziehungsweise Arbeitsbereiche anzulegen. Durch diesen Befehl, auf den sich die meisten Editor-Befehle beziehen, ist es möglich, jedem zusammenhängenden Quelltextteil (Unterprogramme oder Unterprogrammblöcke) einen Arbeitsbereich zuzuordnen. Möchte man dann in der Page 3 etwas ändern oder nachschauen, LISTet der Befehl »/3« nur diesen Bereich und nicht das komplette Listing wie bei dem »/E«-Befehl. Legt man nun die einzelnen Arbeitsbereiche gleich von vornherein so an, daß sie jeweils einen Zeilenbereich von zum Beispiel 5000 Zeilen überdecken, dürfte für die einzelnen Quelltextteile genügend Platz vorhanden sein, so daß beim Durchnumerieren der einzelnen Arbeitsbereiche keine Überlappungen auftreten können. Die Arbeitsbereiche selbst dürfen sich aber durchaus überlappen. So läßt sich zum Beispiel ein Arbeitsbereich von 0 bis 5000, ein zweiter von 10000 bis 15000 und ein dritter von 0 bis 15000 anlegen.

Bei dem Assembler selbst sind bisher keine Fehler bekannt. Deshalb möchte ich an dieser Stelle auf einige Dinge eingehen, mit denen viele Leser Schwierigkeiten hatten. Da wäre zum Beispiel das unmittelbare Erzeugen des Objektcodes auf Diskette mit dem ».OB«-Pseudo-Opcode.

Der Pseudo-Opcode ».OB "filename,P,W"« muß am Anfang des Quelltextes stehen und zwar in der ersten beziehungsweise zweiten Zeile (nach dem ».LI«-Pseudo zur Ausgabe des Assembler-Listings). In dem Zusammenhang sei auch erwähnt, daß es unmöglich ist, den Objektcode und gleichzeitig das Assembler-Listing mit dem Befehl ».LI 2,8,2,"filename,U,W"« auf Diskette zu erzeugen. Der Grund ist der, daß zwei Kanäle zum Schreiben geöffnet werden müßten und das ist nicht möglich. Zu dem ».OB«-Pseudo gehört unmittelbar ein zweiter Pseudo ».EN«, der das mit »filename« gekennzeichnete File schließt. Dazu muß dieser Pseudo am Ende des Quelltextes stehen. Sollten mit dem ».AP« mehrere Quelltexte verkettet werden, muß der ».EN«-Pseudo am Schluß des letzten Quelltextes auftauchen.

Bei der Anwendung von Makros gab es auch einige Schwierigkeiten. Wird zum Beispiel von einem Makro (Ordnung 1) zweimal ein weiteres Makro (Ordnung 2) aufgerufen, meldet Hypra-Ass einen »label twice error«, vorausgesetzt,

im Makro zweiter Ordnung befindet sich ein Label. Zum Beispiel würde folgendes Programm zu einer solchen Fehlermeldung führen:

```
10 -.BA $C000
20 -.MA MAK1
;MAKRODEFINITION 1. ORDNUNG
30 -  ... MAK2
;MAKROAUFRUF 2. ORDNUNG
40 -  ... MAK2
50 -.RT
60 -.MA  MAK2
70 -LBL  NOP
80 -.RT
90 -        ... MAK1
```

Dabei ist Mak1 das Makro 1. Ordnung und Mak2 das Makro 2. Ordnung. Alle Label in Makros zweiter oder dritter Ordnung sind untereinander global. Das heißt, daß in Makros zweiter Ordnung nur einmal das Label mit dem Namen »LBL« definiert werden dürfte.

Im Augenblick wird an einer Erweiterung gearbeitet, die diesen Mißstand beseitigt. Denn gerade beim intensiven Arbeiten mit Makros sind Makros zweiter und sogar dritter Ordnung unabdingbar. Ganz deutlich sieht man dies an dem Artikel »Wichtige Makros für Hypra-Ass« in dieser Ausgabe. Dort wurde ein Makro mit dem Namen »INCW (adresse)« definiert. Würde man die dort stehende 16-Bit-Addition ersetzen durch:

```
        INC   ADRESSE
        BNE   LBL
        INC   ADRESSE+1
LBL
```

könnte dieses Makro von keinem anderen Makro aus zweimal aufgerufen werden, weil durch das Label »LBL« ein »label twice error« erscheinen würde.

Der gleiche Fehler erscheint natürlich auch dann, wenn ein anderes Makro aufgerufen wird, das »LBL« als Label oder Variable benutzt. Denn die Ordnungszahl, die den beiden Labeln »LBL« zugewiesen wird, ist identisch.

Bedingte Assemblierung

Auch mit der bedingten Assemblierung wissen nur die wenigsten etwas anzufangen, obwohl sie gerade im Zusammenhang mit Makros eine große Rolle spielt. Dies soll an einem kleinen Beispiel demonstriert werden:

```
10  -.MA ADW (ADR1,ADR2,SUMME,RETTEN)
20  -.IF RETTEN !=! 1                 ;NUR WENN RETTEN
                                      = 1, WIRD
30  -            PHA                  ;PHA IN DAS MASCHI-
                                      NENPROGRAMM
40  -.EI                              ;ASSEMBLIERT
50  -            CLC
60  -            LDA ADR1
70  -            ADC ADR2
80  -            STA SUMME
90  -            LDA ADR1+1
100 -            ADC ADR2+1
110 -            STA SUMME+1
120 -.IF RETTEN !=! 1                 ;NUR WENN RETTEN
                                      = 1, WIRD
130 -            PLA                  ;PLA IN DAS
                                      MASCHINENPROGRAMM
140 -.EI ;                            ASSEMBLIERT
150 -.RT
```

Dieses Makro addiert (ADR1,ADR1+1)+(ADR2,ADR2+1) und speichert das Ergebnis in den Speicherzellen (SUMME,SUMME+1). ADR1, ADR2 und SUMME können beliebige Speicherzellen oder Variablen sein. Soll der Inhalt des Akkumulators erhalten bleiben, wird für RETTEN eine 1, ansonsten eine beliebige andere Zahl eingegeben. Anhand des Übergabeparameters RETTEN erkennt der Assembler, ob das erzeugte Maschinenprogramm den Maschinenbefehl »PHA« beziehungsweise »PLA« enthalten soll oder nicht. Die Befehle zur bedingten Assemblierung zeigen also einzig und allein eine Wirkung beim Assemblieren. Durch sie wird bestimmt, welche Teile des Quelltextes im erzeugten Maschinenprogramm stehen und welche unter bestimmten Bedingungen übersprungen werden sollen. Schauen Sie sich nun noch einmal die Zeilen 20 und 120 an. Sie finden dort in der bedingten ».IF«-Abfrage den Operator »=«, der wie alle anderen Operatoren auch in Ausrufezeichen einzufassen ist. Durch dieses Ausrufezeichen erkennt Hypra-Ass, daß es sich bei dem Operator »=« um eine Rechenvorschrift aus dem Quelltext heraus handelt. Alle Operatoren können außer bei der bedingten Assemblierung zum Beispiel auch bei der unmittelbaren Adressierung angewendet werden. Ein kleines Beispiel soll die Wirkung dieser Operatoren bei der unmittelbaren Adressierung verdeutlichen:

```
20  -.EQ VARIABLE1 = 10
30  -.EQ VARIABLE2 = 20
40  -        LDA #(VARIABLE1 !O! VARIABLE2)
```

Der Akkumulator wird mit einer Zahl geladen, die mit der Variablen »VARIABLE1« und »VARIABLE2« wie im Basic geORt wird. Das Ergebnis ist folglich 30.

Viele Maschinensprache-Anfänger verwechseln die Befehle zur bedingten Assemblierung mit normalen Basic-Befehlen. Deshalb möchte ich an einem kleinen Beispiel zeigen, was nicht mit der bedingten Assemblierung funktioniert:

```
10  -.BA  $C000
20  -          INC $D020
30  -.GO  20
```

Was nicht funktioniert

Das Programm sollte die Bildschirmrahmenfarbe laufend um 1 incrementieren. Wird der Assembler jedoch gestartet, ersetzt er den Befehl ».GO 20« nicht durch den Befehl »JMP adresse«. Vielmehr versucht er den gesamten Speicher von $C000 bis unendlich mit dem Befehl »INC $D020« zu füllen, denn es fehlt jegliche Abbruchbedingung. Nur mit einer Abbruchbedingung ist der ».GO«-Befehl sinnvoll. Sollte das Maschinenprogramm zehnmal hintereinander den Befehl »INC $D020« enthalten, könnte das so aussehen:

```
10  -.LI 1,3
20  -.BA $C000
30  -.EQ A = 0
40  -.EQ A = A + 1
50  -          INC $D020
60  -.IF A !<! 11
70  -.GO 40
80  -.EI
90  -          JMP $C000
```

Der Assembler überprüft in Zeile 60, ob die Variable »A« kleiner 11 ist. Trifft das zu, wird in Zeile 70 durch den ».GO«-Befehl zur Zeile 40 verzweigt und der Befehl »INC $D020« ein weiteres Mal assembliert. Sobald »A« gleich 10 ist, verzweigt der Assembler in die Zeile 90, übersetzt den Befehl »JMP $C000« und beendet den Assembliervorgang.

Sollte Ihnen der Umgang mit Makros und der bedingten Assemblierung noch nicht klar sein, empfehle ich Ihnen den Artikel »Assemblerbedienung leicht gemacht (2)« in der 64'er Ausgabe 1/85. In diesem Artikel wird ausführlich auf den Gebrauch von Makros und eben der bedingten Assemblierung eingegangen.

(ah)

Wichtige Makros zum Assembler Hypra-Ass

Der beschränkte Befehlssatz des 6502-Prozessors macht Maschinenprogramme unübersichtlich und fehleranfällig. Wir zeigen Ihnen hier, wie sich mit Hypra-Ass der Maschinen-Befehlssatz durch Makros erweitern läßt. Aber nicht nur das ist möglich, Sie können sich auch Ihre eigene, ganz persönliche Sprache basteln.

Besonders dem Maschinensprache-Anfänger dürfte das Wort »Makro« gänzlich unbekannt sein. Denn weder das im C 64 implementierte noch irgendein anderes Basic kennt die Definition eines Makros. Das ist wahrscheinlich auch der Grund dafür, warum Makros nur selten angewendet werden. Sie spielen jedoch gerade beim 6502- beziehungsweise 6510-Prozessor eine wichtige Rolle. Durch Makros läßt sich nämlich der bescheiden ausgefallene Befehlssatz des Prozessors um wichtige Befehle erweitern. Es werden durch Makros prinzipiell keine neuen Maschinenbefehle geschaffen, sind aber Makros einmal definiert, lassen sie sich aufrufen wie ganz normale Maschinenbefehle. Was sind nun eigentlich Makros? Dies soll an einem kleinen Beispiel erklärt werden.

Angenommen, Sie möchten in einem Maschinenprogramm 20 verschiedene 16-Bit Adressen inkrementieren, dann müßte Ihr Programm zwangsläufig zwanzigmal folgende Befehlsfolge enthalten:

```
          INC ADRESSE
          BNE LBL
          INC ADRESSE+1
     LBL
```

Diese 20 Befehlsfolgen machen aber das Programm unübersichtlich und vor allen Dingen fehleranfällig. Genau so gut ließe sich auch ein Makro mit dem Namen »INCW (adresse)« für INCWord definieren, das dann anstelle der Befehlsfolgen 20mal im Quelltext erscheint. Das Makro selbst würde wie folgt aussehen:

```
10 -.MA INCW (ADRESSE)
20 -       INC ADRESSE
30 -       BNE LBL
40 -       INC ADRESSE+1
50 -LBL
60 -.RT
```

Aufgerufen wird das Makro im Quelltext nun durch den neuen Befehl »... INCW (adresse)«.
Gefolgt von dem Makronamen und in Klammern den Übergabeparametern, die durch Kommata getrennt werden, leitet der ».MA«-Pseudo-Opcode die Definition eines Makros ein. Dies geschieht in Zeile 10. Der Pseudo-Opcode ».RT« in Zeile 60 schließt die Definition des Makros ab. Alle im Makro stehenden Label sind lokal. Das heißt, daß dem Programm außerhalb des Makros die internen Label unbekannt sind. Würde dies nicht so sein, dann würde der Assembler den zweiten Makroaufruf mit der Fehlermeldung »label twice error« ahnden. Was macht der Assembler, wenn er auf einen Makroaufruf stößt? Er assembliert in den Objektcode, wie man das erzeugte Maschinenprogramm auch nennt, die Befehlsfolgen, die im Makro definiert wurden. Das heißt, daß letztendlich im erzeugten Maschinenprogramm wieder zwanzigmal, um bei dem Beispiel zu bleiben, die oben stehenden Befehlsfolgen auftauchen.
Im Listing sind die wichtigsten Makros aufgeführt. Neben den »Befehlserweiterungen« ist noch eine interessante Gruppe von Makros definiert worden, die die strukturierte Programmierung durch »Repeat... Until«- und »While... Endwhile«-Schleifen unterstützt. Zu beachten ist jedoch, daß die Schleifen nicht verschachtelt werden dürfen. Schleifenkonstruktionen wie

```
REPEAT
 .
 .
    REPEAT
     .
     .
    UNTIL
UNTIL
```

sind veboten. Die einzelnen Makros haben folgende Wirkung:
TXY: Das Y-Register wird mit dem Inhalt des X-Registers geladen.
TYX: Das X-Register wird mit dem Inhalt des Y-Registers geladen.
PHX: Das X-Register wird auf dem Stack abgelegt.
PHY: Das Y-Register wird auf dem Stack abgelegt.
PLX: Das X-Register wird vom Stack geholt.
PLY: Das Y-Register wird vom Stack geholt.
Die folgenden vier Makros definieren einen Userstack, der an eine beliebige Stelle gelegt werden kann. Dazu muß im Hauptprogramm eine globale Variable mit dem Namen »USER« in der Zeropage angelegt werden. Anschließend muß in die Adresse, die die Variable repräsentiert, die Startadresse des Stacks geschrieben werden. Das könnte so aussehen:

```
10 -.GL USER = 3
20 -       LDA #0          ;LO-BYTE
STARTADRESSE USERSTACK
30 -       STA USER
40 -       LDA #$C0        ;HI-BYTE
STARTADRESSE USERSTACK
50 -       STA USER+1
```

Hier wurde ein Userstack angelegt, der bei Adresse $C000 beginnt. Der Stackpointer, also der Zeiger, der auf die aktuelle Stackadresse zeigt, steht in der Zeropage in den Speicherzellen 3 und 4.
PUSHA: Der Inhalt des Akkumulators wird auf dem Userstack abgelegt.
PUSHAY: Der Inhalt des Akkumulators und der Inhalt des Y-Registers werden auf dem Userstack abgelegt.
PULLA: Der Akkumulator wird vom Userstack geholt.
PULLAY: Der Akkumulator und das Y-Register werden vom Userstack geholt.
ADW (adresse): 16-Bit Addition. Der Inhalt einer beliebigen Adresse wird zum Inhalt des Akkumulators (Low-Byte) und zum Inhalt des Y-Registers (High-Byte) addiert. Das Ergebnis steht anschließend im Akkumulator (Low-Byte) und im Y-Register (High-Byte).
ADMW (adr1,adr2,summe): 16-Bit Addition. Der Inhalt von adr1 und adr1+1 wird zum Inhalt der Adresse adr2 und adr2+1 addiert und das Ergebnis in der Adresse summe und summe+1 abgelegt.
SBCW (adresse): 16-Bit Subtraktion. Der Inhalt von adresse und adresse+1 wird vom Inhalt des Akkumulators (Low-Byte) und vom Inhalt des Y-Registers (High-Byte) abgezogen. Das

Ergebnis steht anschließend im Akkumulator (Low-Byte) und im Y-Register (High-Byte).
SBCMW (adr1,adr2,diff): 16-Bit Subtraktion. Vom Inhalt adr1 und adr1+1 wird der Inhalt von adr2 und adr2+1 abgezogen. Das Ergebnis wird in der Adresse diff und diff+1 abgelegt.
INCW (adresse): Der Inhalt von adresse und adresse+1 wird inkrementiert. Das Ergebnis steht in adresse und adresse+1.
DECW (adresse): Der Inhalt von adresse und adresse+1 wird dekrementiert. Das Ergebnis steht in adresse und adresse+1.
LDAY (adresse): Der Akkumulator wird mit dem Inhalt von adresse und das Y-Register mit dem Inhalt von adresse+1 geladen.
STAY (adresse): Der Inhalt des Akkumulators wird nach adresse und der Inhalt des Y-Registers nach adresse+1 geschrieben.
LDAYI (wert): Der Akkumulator und das Y-Register wird mit »wert« unmittelbar geladen. Dabei steht das Low-Byte im Akkumulator und das High-Byte im Y-Register.
Die folgenden Makros unterstützen die strukturierte Programmierung.

REPEAT, EXITREPEAT, UNTIL (übergabe,bedingung): Die Schleife wird so lange fortgesetzt, bis die Speicherzelle »übergabe« den Wert »bedingung« enthält. Beispiel:

```
10 -          LDX #255
20 -          ... REPEAT
30 -          DEX
40 -          STX $FB
50 -          ... UNTIL ($FB,0)
```

Das X-Register wird solange dekrementiert, bis es den Wert »0« enthält.
WHILE (übergabe,bedingung) , EXITWHILE, ENDWHILE: Die Schleife wird solange fortgesetzt, bis der Inhalt der Speicherzelle »übergabe« gleich »bedingung« ist. Beispiel:

```
10 -          LDX #255
20 -          ... WHILE ($FB,0)
30 -          DEX
40 -          STX $FB
50 -          ... ENDWHILE
```

Solange der Inhalt der Speicherzelle $FB ungleich Null ist, wird das X-Register dekrementiert.

(ah)

```
READY.

10   -;**********************************
20   -;*   WEITERE VERSCHIEBEBEFEHLE   *
30   -;**********************************
40   -;
50   -; X -> Y
60   -.MA TXY
70   -             PHA
80   -             TXA
90   -             TAY
100  -             PLA
110  -.RT
120  -;
130  -; Y -> X
140  -.MA TYX
150  -             PHA
160  -             TYA
170  -             TAX
180  -             PLA
190  -.RT
200  -;
210  -; X-REGISTER AUF DEN STACK
220  -.MA PHX
230  -.EQ RETTEN = $FC
240  -             STA RETTEN
250  -             TXA
260  -             PHA
270  -             LDA RETTEN
280  -.RT
290  -;
300  -; Y-REGISTER AUF DEN STACK
310  -.MA PHY
320  -.EQ RETTEN = $FC
330  -             STA RETTEN
340  -             TYA
350  -             PHA
360  -             LDA RETTEN
370  -.RT
380  -;
390  -; X-REGISTER VOM STACK HOLEN
400  -.MA PLX
410  -.EQ RETTEN = $FC
420  -             STA RETTEN
430  -             PLA
440  -             TAX
450  -             LDA RETTEN
460  -.RT
470  -;
480  -; Y-REGISTER VOM STACK HOLEN
490  -.MA PLY
500  -.EQ RETTEN = $FC
510  -             STA RETTEN
520  -             PLA
530  -             TAY
540  -             LDA RETTEN
550  -.RT
560  -;
570  -; DEN AKKU AUF USERSTACK
580  -.MA PUSHA
590  -             ... PHY
600  -             LDY #0
610  -             STA (USER),Y
620  -             ... DECW(USER)
630  -             ... PLY
640  -.RT
650  -;
660  -;AKKU UND Y-REGISTER AUF USERSTACK
670  -.MA PUSHAY
680  -             PHA
690  -             TYA
700  -             ... PUSHA
710  -             PLA
720  -             ... PUSHA
730  -.RT
740  -;
750  -; AKKU VON USERSTACK
760  -.MA PULLA
770  -             ... INCW(USER)
780  -             ... PHY
790  -             LDY #0
800  -             LDA (USER),Y
810  -             ... PLY
820  -.RT
830  -;
840  -;AKKU UND Y-REGISTER VON USERSTACK
850  -.MA PULLAY
860  -             ... PULLA
870  -             TAY
880  -             ... PULLA
890  -.RT
900  -;
910  -;**********************************
920  -;*       16-BIT BEFEHLE           *
930  -;**********************************
940  -;
950  -; A/Y + ADRESSE = A/Y AKKU=LO
960  -.MA ADW (ADRESSE)
970  -             CLC
980  -             ADC ADRESSE
990  -             PHA
1000 -             TYA
1010 -             ADC ADRESSE+1
1020 -             TAY
1030 -             PLA
1040 -.RT
1050 -;
1060 -; ADIERE ADR1 + ADR2 = SUMME
1070 -.MA ADMW (ADR1,ADR2,SUMME)
1080 -             PHA
1090 -             CLC
1100 -             LDA ADR1
1110 -             ADC ADR2
1120 -             STA SUMME
1130 -             LDA ADR1+1
1140 -             ADC ADR2+1
1150 -             STA SUMME+1
1160 -             PLA
1170 -.RT
1180 -;
1190 -; A/Y - ADRESSE = A/Y AKKU=LO
1200 -.MA SBCW (ADRESS)
1210 -             SEC
1220 -             SBC ADRESSE
1230 -             PHA
1240 -             TYA
1250 -             SBC ADRESSE+1
1260 -             TAY
1270 -             PLA
1280 -.RT
1290 -;
1300 -; ADR1 - ADR2 = DIFFERENZ
1310 -.MA SBCMW (ADR1,ADR2,DIFF)
1320 -             PHA
1330 -             SEC
1340 -             LDA ADR1
1350 -             SBC ADR2
1360 -             STA DIFF
1370 -             LDA ADR1+1
1380 -             SBC ADR2+1
1390 -             STA DIFF+1
1400 -             PLA
1410 -.RT
1420 -;
1430 -; ADRESSE = ADRESSE + 1
1440 -.MA INCW (ADRESSE)
1450 -             PHA
1460 -             LDA ADRESSE
1470 -             CLC
1480 -             ADC #1
1490 -             STA ADRESSE
1500 -             LDA ADRESSE+1
1510 -             ADC #0
1520 -             STA ADRESSE+1
1530 -             PLA
1540 -.RT
1550 -;
1560 -; ADRESSE = ADRESSE - 1
1570 -.MA DECW (ADRESSE)
1580 -             PHA
1590 -             LDA ADRESSE
1600 -             SEC
1610 -             SBC #1
1620 -             STA ADRESSE
1630 -             LDA ADRESSE+1
1640 -             SBC #0
1650 -             STA ADRESSE+1
1660 -             PLA
1670 -.RT
1680 -;
1690 -; ADRESSE -> A/Y AKKU=LO
1700 -.MA LDAY (ADRESSE)
1710 -             LDY ADRESSE+1
1720 -             LDA ADRESSE
1730 -.RT
1740 -;
1750 -; A/Y -> ADRESSE AKKU=LO
1760 -.MA STAY (ADRESSE)
1770 -             STA ADRESSE
1780 -             STY ADRESSE+1
1790 -.RT
1800 -;
1810 -; WERT=16BIT -> A/Y
1820 -.MA LDAYI (WERT)
1830 -             LDA #<(WERT)
1840 -             LDY #>(WERT)
1850 -.RT
1860 -;**********************************
1870 -;*  BEFEHLE ZUR STRUKTURIERTEN    *
1880 -;*       PROGRAMMIERUNG           *
1890 -;**********************************
1900 -;
1910 -.MA REPEAT
1920 -ACE1         .GL ACE0=ACE1
1930 -.RT
1940 -;
1950 -.MA EXITREPEAT
1960 -             JMP BCE0
1970 -.RT
1980 -;
1990 -.MA UNTIL (UEBERGABE,BEDINGUNG)
2000 -             PHA
2010 -             LDA UEBERGABE
2020 -             CMP #BEDINGUNG
2030 -             BEQ LBL1
2040 -             PLA
2050 -             JMP ACE0
2060 -LBL1         PLA
2070 -.GL BCE0=LBL1
2080 -.RT
2090 -;
2100 -.MA WHILE (UEBERGABE,BEDINGUNG)
2110 -CCE0         .GL CCE0=CCE0
2120 -             PHA
2130 -             LDA UEBERGABE
2140 -             CMP #BEDINGUNG
2150 -             BNE LBL1
2160 -             JMP CCE1
2170 -LBL1         PLA
2180 -.RT
2190 -;
2200 -.MA EXITWHILE
2210 -             PHA
2220 -             JMP CCE1
2230 -.RT
2240 -;
2250 -.MA ENDWHILE
2260 -             JMP CCE0
2270 -CCE1         .GL CCE1=CCE1
2280 -             PLA
2290 -.RT
```

Listing. Die wichtigsten Makros zum Assembler »Hypra-Ass«

Tips & Tricks ausführlich erklärt

Die folgenden Programmbeispiele sollen vor allem dem Anfänger den Einstieg in die Maschinensprache des C 64 erleichtern.

Zu ihrem Verständnis sollte man wenigstens ungefähr mit dem Befehlssatz der 6510-CPU und mit der Speicherorganisation des C 64 vertraut sein. Die Beispiele stammen aus den verschiedensten Anwendungsgebieten. Ihnen allen gemeinsam ist:

- eine überschaubare Kürze
- Formulierung als Hypra-Ass-Quelltext
- eine ausführliche Beschreibung der Wirkungsweise.

1. Ein allererster Gehversuch mit Hypra-Ass

Zu diesem und zu allen folgenden Programmen benötigen Sie den Hypra-Ass. Er wird wie ein Basic-Programm geladen und mit RUN initialisiert. Jedes weitere RUN startet jetzt einen Assemberlauf, ist aber zunächst noch wirkungslos, da noch nichts im Textspeicher des Assemblers steht. Geben Sie nun folgende vier Zeilen ein:

```
10 - .BA $C000
20 - LDA #1
30 - STA $400
40 - RTS
```

Dabei dürfen die Minus-Zeichen nach den Zeilennummern nicht vergessen werden (Eigenart von Hypra-Ass). Auch das Leerzeichen nach den Minus-Zeichen ist wichtig. Sie werden feststellen, daß der Assember die Zeilen nach Drücken der RETURN-Taste formatiert. Listen Sie die vier Zeilen auch einmal probeweise mit

LIST (unformatiert) und
/E (formatiert)

Das Programm ist schnell erklärt:

Mit .BA $C000 wird dem Assembler die Startadresse (BAsisadresse) des Programms mitgeteilt. .BA ist ein Pseudobefehl. Ein solcher Befehl steuert die Arbeitsweise des Assemblers, bewirkt aber keine Erzeugung eines Maschinenbefehls.

LDA #1 lädt den Akkumulator mit 1, dem Bildschirmcode des Buchstabens A.

STA $400 speichert den Akkumulatorinhalt, also die 1 an die Speicherstelle $400. $400 ist die Startadresse des Bildschirm-RAMs und entspricht der linken oberen Bildschirmecke. Dort muß also ein »A« erscheinen.

RTS (ReTurn from Subroutine) bedeutet Rücksprung aus einem Unterprogramm. Mit RTS müssen Programme abgeschlossen sein, die mit JSR (Jump to SubRoutine) oder SYS (von Basic aus) aufgerufen werden. Dies dürfte für die überwiegende Mehrheit aller Maschinenprogramme der Fall sein. Ausnahmen sind:

Programme., die von einem Monitor aus gestartet werden. Sie sollten mit einem BRK (Break) abgeschlossen sein.

Programme, die durch Interrupts aktiviert werden, werden normalerweise durch RTI (ReTurn from Interrupt) abgeschlossen.

Mit RUN wird der Assembler gestartet. Er erzeugt ein 6 Byte kurzes Maschinenprogramm ab $C000 und gibt in einer Abschlußmeldung den belegten Speicherbereich zusammen mit der Assemblierzeit bekannt. Das Maschinenprogramm kann nun mit

```
SYS $C000 oder SYS 12 * 4096
```

gestartet werden. (Wenn Hypra-Ass aktiv ist, versteht der C 64 auch Hex-Zahlen). Es müßte ein »A« in der linken oberen Bildschirmecke erscheinen. Sollte das nicht der Fall sein, so kann das zwei Ursachen haben:

1. Das »A« wurde nach oben weggescrollt, weil Sie den SYS-Befehl zu weit unten auf dem Bildschrim eingetippt haben.
2. Sie besitzen eine alte Version des C 64, bei der das Farb-RAM mit der Hintergrundfarbe vorbesetzt wird.

Im zweiten Fall ergänzen Sie das Programm durch:

```
35 - STA $D800
```

Dieser Befehl speichert die immer noch im Akkumulator (im folgenden nur Akku genannt) stehende 1 an die Startadresse des Farb-RAMs. Dadurch erscheint das »A« in weißer Farbe.

Ergänzen Sie Ihr kleines Programm einmal durch den Pseudobefehl

```
5 - .LI 1,3,0
```

und assemblieren Sie mit RUN. Der Assembler erzeugt jetzt ein Listing, dessen Zeilen von links nach rechts wie folgt aufgebaut sind:

- Speicheradresse des folgenden Maschinenbefehls
- Der Code des Maschinenbefehls. Da es Maschinenbefehle mit ein, zwei oder drei Byte gibt, sind diese Einträge unterschiedlich lang.
- Ein Doppelpunkt und die ursprüngliche Quelltextzeile. Bei Pseudobefehlen, die ja keinen Code erzeugen, entfällt der Teil bis einschließlich zum Doppelpunkt.

Das Assemblerlisting ist bei der Fehlersuche mit einem Monitor nützlich, da es zu jedem Maschinenbefehl seine Adresse enthält. Mit dem Pseudobefehl:

```
.li 1,4,0
```

erhält man ein Druckerlisting. Angenehm dabei ist, daß es bei ausgeschaltetem Drucker automatisch auf den Bildschrim umgeleitet wird.

Die abgedruckten Listings enthalten allerdings keine Speicheradressen mit zugehörigem Maschinencode. Diese Information ist zum Studieren der Programme uninteressant und zum Eintippen der Listings nicht erforderlich. Die Listings wurden mit dem Editorbefehl /E (formatiertes Listen) gewonnen, nachdem vorher die Ausgabe durch

```
OPEN 4,4:CMD 4
```

auf den Drucker umgeleitet wurde.

2. Eine einfache Programmschleife

Das folgende kleine Programm (Listing 1) schreibt 240mal den Buchstaben »A« auf den Bildschirm. Zur Arbeitsweise: Akku A wird wieder mit dem Bildschirmcode des Buchstabens A geladen. Das X-Register übernimmt zwei Aufgaben: Es zählt Schleifendurchläufe und liefert Werte zur Adreßverschiebung. X wird mit 0 vorbesetzt. In Zeile 1280 ist

```
100  -;--------------------------------
110  -; PROGRAMMSCHLEIFE
120  -; 240 MAL "A"
130  -; AUF BILDSCHIRM SCHREIBEN
140  -;--------------------------------
150  -          .BA $C000     ;PROGRAMM-STARTADRESSE
160  -          LDA #1        ;BILDSCHIRMCODE VON "A"
170  -          LDX #0        ;SCHLEIFENZAEHLER
180  -LOOP      STA $400,X    ;ZEICHEN AUF BILDSCHIRM SCHREIBEN
190  -          INX           ;X HOCHZAEHLEN
200  -          CPX #240      ;X MIT 240 VERGLEICHEN
210  -          BNE LOOP      ;FALLS UNGLEICH, GEHE NACH "LOOP"
220  -          RTS           ;FALLS X=240, DANN PROGRAMMENDE

READY.
```

Listing 1. Programmschleife

LOOP ein Label (auch SYMBOL oder Sprungmarke). LOOP steht für die Adresse des STA-Befehls, die man an anderen Programmstellen durch den Namen LOOP ansprechen kann, ohne daß man den Wert dieser Adresse kennt. Den Assemblerprogrammierer interessieren absolute Adressen normalerweise auch gar nicht, es sei denn, es handelt sich um systemspezifische Adressen, wie zum Beispiel die Adressen der Video-Chip-Register. Eine derartige Zuordnung einer Programmadresse zu einem Label nennt man implizite Definition.

Die Zieladresse des STA-Befehls in Zeile 180 ergibt sich durch die Summe aus $400 und X. Da X am Anfang 0 gesetzt worden ist, wird also eine 1 an die Stelle $400 gespeichert. INX erhöht X um Eins. CPX #240 vergleicht X mit der Zahl 240. Bei Ungleichheit wird das Zero-Flag im Statusregister auf 0 gesetzt. Auf dieses Zero-Flag bezieht sich dann der bedingte Sprung BNE LOOP. BNE springt dann, wenn der vorige Vergleich Ungleichheit ergeben hat. (Daher auch der Name BNE = Branch if Not Equal = verzweige, wenn ungleich). Beim nächsten Schleifendurchlauf wird die 1 aus dem Akku an die Adresse $4012 gespeichert. X wird solange inkrementiert, bis 240 erreicht ist. In diesem Fall springt BNE nicht und das Programm endet mit RTS. Bei den 240 Schleifendurchläufen werden nacheinander die Adressen $400 bis $400+239 angesprochen. Programmschleifen wie diese benützt man oft zum Löschen eines Speicherbereichs. (Akku mit 0 vorbesetzt.)

3. Blockverschiebung (maximal 255 Byte)

Das Programm (Listing 2) arbeitet mit einer ähnlichen Schleife wie das vorige. In den Zeilen 150, 160 und 170 werden Label explizit definiert. Dies geschieht mit dem Pseudobefehl .EQ.

Die explizite Definition eines Labels ist praktisch dasselbe wie die Zuweisung eines Wertes an eine Variable. Hier werden die Anfangsadressen des ursprünglichen Blocks und des verschobenen Blocks sowie die Blocklänge definiert. X läuft hier rückwärts von LAENGE bis 0. Dadurch kann der CPX-Befehl eingespart werden. DEX setzt nämlich automatisch das Zero-Flag, wenn nach dem Dekrement X den Wert 0 hat. BNE LOOP springt also nur solange nach LOOP, solange X größer als 0 ist. Mit den vorliegenden Werten für QUELLE, ZIEL und LAENGE kopiert das Programm die Bildschirmzeilen 1 bis 6 auf die Zeilen 7 bis 12.

Programme zur Blockverschiebung wie dieses oder zur Blockfüllung wie das vorige sind nur für Blocklängen bis maximal 255 Byte geeignet, da das X-Register nur 8 Bit lang ist. Wenn man größere Speicherbereiche auf diese Weise verarbeiten will, muß man mehr Aufwand treiben.

4. Blockverschiebung (ohne Einschränkungen)

Das Programm (Listing 3) ist sicher nicht die kürzeste Lösung des Problems, es demonstriert dafür aber ohne verwirrende Tricks die Adressierungsart »Indirekt Indiziert«.

Beispiel: `LDA (ZEIGER),Y`

Bei dieser Adressierungsart enthalten zwei aufeinanderfolgende Speicherstellen der Zero-Page eine Adresse in der üblichen Reihenfolge Low-Byte – High-Byte. Im Programm wird nicht diese Adresse selbst angegeben, sondern die Adresse der ersten der beiden Zero-Page-Speicherstellen (hier ZEIGER genannt). Diese Technik nennt man indirekte Adressierung, was im Assemblertext durch die runden Klammern um die Zero-Page-Adresse zum Ausdruck kommt. Zu der aus der Zero-Page stammenden Adresse wird noch Y addiert, daher »indiziert«. Da man diese zusätzliche Indizierung oft nicht braucht, setzt man das Y-Register vorher auf 0.

Das Programm verwendet für den Blocktransfer zwei Zeiger (= Zero-Page-Speicherstellenpaare). Sie werden mit der Startadresse des Quell- beziehungsweise des Zielblocks initialisiert und nach jedem Byte-Transfer hochgezählt, bis der Zeiger in den Quellblock (ZEIGER1) das Ende des Quellblocks (Adresse BIS) erreicht hat.

Das Inkrementieren eines 16-Bit Wertes verläuft nach dem Schema:

```
        INC ZEIGER   ;Low-Byte inkrementieren
        BNE Weiter   ;falls ungleich 0, dann fertig
        INC ZEIGER+1;Übertrag ins High-Byte
WEITER (Programmfortsetzung)
```

Spezifisch für den Hypra-Ass ist, daß man mit
<(Adresse) beziehungsweise >(Adresse)
das Low- bezeihungsweise High-Byte einer Adresse (beziehungsweise eines Labels) gezielt ansprechen kann. Von dieser Möglichkeit wird im Programm häufig Gebrauch gemacht. So bedeutet zum Beispiel:

```
LDA #<     (VON)
```

Lade den Akkumulator mit dem Low-Byte des Wertes VON.

Mit den im Program definierten Adressen VON, BIS und ZIEL kopiert das Programm einen Teilbereich aus dem Basic-

```
100  -;--------------------------------
110  -; BLOCKVERSCHIEBUNG
120  -; (MAXIMAL 255 BYTE)
130  -;--------------------------------
140  -           .BA $C000      ;PROGRAMMSTART
150  -           .EQ QUELLE=$400;BLOCKSTART
160  -           .EQ ZIEL=$400+240
170  -           .EQ LAENGE=240 ;BLOCKLAENGE
180  -; X LAEUFT RUECKWAERTS VON LAENGE BIS 1
190  -           LDX #LAENGE
200  -LOOP       LDA QUELLE-1,X
210  -           STA ZIEL-1,X
220  -           DEX            ;X:=X-1
230  -           BNE LOOP       ;FALLS X<>0, NACH LOOP
240  -           RTS            ;SONST ENDE
READY.
```

Listing 2. Blockverschiebung


```
100  -;--------------------------------
110  -; BLOCKVERSCHIEBUNG
120  -; OHNE EINSCHRAENKUNGEN
130  -;--------------------------------
140  -           .BA $C000      ;PROGRAMMSTART
150  -           .EQ VON=$A09E
160  -           .EQ BIS=$A327
170  -           .EQ ZIEL=$400
180  -           .EQ ZEIGER1=$FB
190  -           .EQ ZEIGER2=$FD
200  -;
210  -           LDA #<(VON)
220  -           STA ZEIGER1
230  -           LDA #>(VON)
240  -           STA ZEIGER1+1
250  -           LDA #<(ZIEL)
260  -           STA ZEIGER2
270  -           LDA #>(ZIEL)
280  -           STA ZEIGER2+1
290  -           LDY #0
300  -LOOP       LDA (ZEIGER1),Y
310  -           STA (ZEIGER2),Y
320  -; ZEIGER1 MIT "BIS" VERGLEICHEN
330  -           LDA ZEIGER1
340  -           CMP #<(BIS)
350  -           BNE WEITER
360  -; LOW-BYTES STIMMEN UEBEREIN, HIGH-BYTES VERGLEICHEN
370  -           LDA ZEIGER1+1
380  -           CMP #>(BIS)
390  -           BEQ ENDE
400  -;          BEIDE ZEIGER INKREMENTIEREN
410  -WEITER     INC ZEIGER1
420  -           BNE WEITER2
430  -           INC ZEIGER1+1
440  -WEITER2    INC ZEIGER2
450  -           BNE LOOP
460  -           INC ZEIGER2+1
470  -           JMP LOOP
480  -ENDE       RTS
READY.
```

Listing 3. Blockverschiebung ohne Einschränkung

Interpreter direkt auf den Bildschrim. Im Groß-/Kleinschrift-modus (Commodore-Shift drücken) kann man dann Basic-Schlüsselwörter sowie Texte von Fehlermeldungen lesen.

5. Verwendung von Betriebssystem-Funktionen und Mechanismen zur Parameterübergabe

Ein Betriebssystem ist unter anderem dazu da, Standarddienste wie Ein- und Ausgabe zur Verfügung zu stellen, damit diese nicht jedesmal mühsam und fehleranfällig neu programmiert werden müssen. Die Standardfunktionen des Betriebssystems (oft »Kernel« genannt) sind im Programmierhandbuch von Commodore hinreichend erläutert. Viele weitere nützliche Routinen findet man beim Studium eines kommentierten ROM-Listings.

Die Parameterübergabe an Maschinensprache-Unterprogramme gestaltet sich leider nicht so systematisch wie bei den meisten höheren Programmiersprachen. Es werden mehrere Möglichkeiten bunt gemischt angewendet.

1. Man schreibt Parameter in vereinbarte Speicherstellen. Aus diesen holt sich dann das aufgerufene Programm die Parameter.
2. Wenn nur Ein- bis Drei-Byte-Parameter benötigt werden, kann man diese auch in den Registern A, X und Y übergeben. Auf diese Weise werden die meisten Kernel-Funktionen mit Parametern versorgt.
   Dieser Mechanismus steht übrigens auch von Basic aus zur Verfügung: Man schreibt Registerparameter per POKE an speziell dafür vorgesehene Speicherstellen:

Akku A	780 ($30C)
Index X	781 ($30D)
Index Y	782 ($30E)
Status-Register	783 ($30f)

   Das Maschinenprogramm (Listing 4) wird nur mit SYS aufgerufen. Der Basic-Interpreter besetzt erst die Register mit den Inhalten dieser Speicherstellen und bringt dann in das Unterprogramm. Nach der Rückkehr werden die (neuen) Registerinhalte wieder in denselben Speicherstellen abgelegt, wo sie für eine eventuelle Inspektion durch das Basic-Programm zur Verfügung stehen.

```
100  -;---------------------------------
110  -; VERWENDUNG DER BETRIEBSSYSTEM-
120  -; ROUTINEN GETIN UND CHROUT
130  -;---------------------------------
140  -          .BA $C000
150  -          .EQ GETIN=$FFE4
160  -          .EQ CHROUT=$FFD2
170  -          .EQ ZAEHLER=$FE
180  -          .EQ MAXLEN=10   ;ZEILENLAENGE
190  -          .EQ PROMPT=63   ;"?"
200  -          .EQ ESCAPE=88   ;FLUCHTSYMBOL "X"
210  -          .EQ SPACE=32
220  -          .EQ CR=13       ;CARRIAGE RETURN
230  -;
240  -NEWLINE   LDA #MAXLEN     ;ZAEHLER INITIALISIEREN
250  -          STA ZAEHLER
260  -          LDA #CR
270  -          JSR CHROUT      ;ZEILENVORSCHUB
280  -          LDA #PROMPT
290  -          JSR CHROUT      ;PROMPT-ZEICHEN AUSGEBEN
300  -          LDA #SPACE
310  -          JSR CHROUT
320  -WAIT      JSR GETIN       ;AUF EINGABE WARTEN
330  -          CMP #0
340  -          BEQ WAIT
350  -          CMP #ESCAPE     ;BEI ESCAPE-ZEICHEN PROGRAMMENDE
360  -          BEQ ENDE
370  -          CMP #CR         ;BEI CR NEUE ZEILE
380  -          BEQ NEWLINE
390  -          JSR CHROUT      ;EINGABEZEICHEN WIEDER AUSGEBEN
400  -          DEC ZAEHLER
410  -          BNE WAIT        ;NAECHSTE EINGABE
420  -          BEQ NEWLINE     ;ZEILENLAENGE ERREICHT
430  -ENDE      RTS

READY.
```

Listing 4. Verwendung von Betriebssystemroutinen

3. Man kann Parameter auch über den Stack übergeben. Diese Methode ist wegen des kleinen Stackbereichs der 6510-CPU (256 Byte) nur bedingt brauchbar und wird deshalb auch kaum praktiziert.
4. Durch geschickte Verwendung von Unterprogrammen in Basic-ROM kann man Parameter direkt hinter den SYS-Befehl schreiben. Diese Methode ist
   - komfortabel, weil keine umständlichen POKEs nötig sind
   - schnell, weil der Interpreter weniger zu tun hat
   - flexibel, weil als Parameter auch ganze arithmetische oder Stringausdrücke geschrieben werden können.

   Diese Methode wird in den Programm-Listings 5 bis 8 verwendet.

Das folgende Programm (Listing 4) nutzt die Funktionen GETIN und CHROUT. GETIN liefert den ASCII-Code einer gedrückten Taste im Akku. Falls keine Taste gedrückt wurde, wird 0 zurückgegeben. GETIN entspricht damit genau dem GET-Befehl in Basic.

CHROUT gibt ein Zeichen, dessen ASCII-Code im Akku stehen muß, auf dem Bildschrim aus. Es entspricht dem Basic-Befehl (man beachte das Semikolon):

```
PRINT CHR$(A);
```

Das Programm gibt einen Prompt aus und erwartet anschließend Eingaben. Unter einem Prompt versteht man ein (beliebig zu vereinbarendes) Zeichen am linken Bildschirmrand, das dem Benutzer mitteilt, daß Eingaben von ihm erwartet werden. Bei interaktiven Programmen (wie zum Beispiel Monitore, Editore) sind Prompts sehr nützlich, da der Benutzer daran eindeutig erkennen kann, in welchem Programm er gerade ist. Das vorliegende Programm gibt die Eingabezeichen sofort wieder aus, ohne sie weiter zu verarbeiten. Nach maximal zehn Zeichen wird automatisch ein Zeilenvorschub ausgeführt und ein weiterer Prompt ausgegeben. Das Programm ist eine Endlosschleife, die man mit der Eingabe von »X« verlassen kann.

6. Verwendung von Interpreter-Routinen zur Parameterübergabe

Diese Interpreter-Routinen werden in den folgenden Programmbeispielen eingesetzt:

CHKKOM ($AEFD)	liest aus dem laufenden Basic-Text ein Komma. Steht an der aktuellen Stelle kein Komma, wird das Programm mit SYNTAX ERROR abgebrochen. Kommata sind nötig, um Parameter voneinander abzugrenzen.
FRMNUM ($AD8A)	wertet einen beliebigen arithmetischen Ausdruck aus. Das Ergebnis wird im Fließkomma-Akkumulator 1 (kurz FAC) abgelegt. Der FAC besteht aus den Speicherstellen $61-$66. Die Bedeutung der einzelnen Byte ist hier nicht relevant.
GETADR ($B7F7)	wandelt den Inhalt des FAC in ein 2-Byte-Integer-Format um, sofern diese Zahl im Bereich 0 ... 65535 liegt. Ansonsten wird ein ILLEGAL QUANTITY ERROR ausgegeben. Die Integerzahl steht in den Speicherstellen $14/$15 und zusätzlich im Registerpaar Y/A. Mit der Kombination FRMNUM und GETADR kann man also 16-Bit-Größen aus Basic-Programmen übernehmen.
XBYTE ($B79E)	wertet ebenfalls arithmetische Ausdrücke aus und wandelt das Ergebnis in 8-Bit-Integerformat, sofern es im Bereich 0 ... 255 liegt. Das Byte-Ergebnis wird im X-Register übergeben.
USR ( )	ist eine Basic-Funktion, mit der man Werte von Maschinenprogrammen an Basic zurückgeben kann. USR wertet einen in Klammern stehenden

Ausdruck aus und übergibt ihn in den FAC. Es wird ein Maschinenprogramm aufgerufen, dessen Startadresse in $311/$312 steht. (USR-Vektor). Das Maschinenprogramm kann dann im FAC einen Wert an Basic zurückgeben.

Das folgende Listing (Listing 5) ist der Programmkopf zu den vier nachfolgenden Beispielen. Diese können mit dem Kopf zusammen assembliert werden. Der Kopf enthält eine Sprungliste. Dadurch werden Einsprungstellen ($C000, $C003, etc.) für die vier aufgeführten Programme fixiert, unabhängig davon, wo die Programme dann später tatsächlich im Speicher stehen. Diese Technik ist zum Beispiel

```
100  -;================================
110  -; EINIGE ALLGEMEIN NUETZLICHE
120  -; MASCHINENSPRACHE-UNTERPROGRAMME
130  -; FUER DEN AUFRUF DURCH
140  -; BASIC-PROGRAMME
150  -;
160  -; EINFACHE PARAMETERUEBERGABE:
170  -; SYS STARTADRESSE,PARAMETERLISTE
180  -;================================
190  -;
200  -          .BA $C000
210  -; EINSPRUNGPUNKTE UND UNTERROUTINEN
220  -; DES BASIC-INTERPRETERS
230  -          .EQ CHKKOM=$AEFD;PRUEFT AUF KOMMA
240  -          .EQ FRMNUM=$AD8A;BERECHNET NUMERISCHEN AUSDRUCK IN FAC
250  -          .EQ GETADR=$B7F7;WANDELT FAC IN INTEGERFORMAT ($14/$15)
260  -          .EQ XBYTE=$B79E;HOLT BYTE-WERT NACH X
270  -          .EQ PLOT=$FFF0 ;CURSOR SETZEN
280  -          .EQ PRINT=$AAA0;BASIC-PRINT
290  -          .EQ SETLFS=$FFBA;FILEPARAMETER SETZEN
300  -          .EQ SAVE=$FFD8
310  -;
320  -; SPRUNGLISTE
330  -;
340  -          JMP PRINTAT
350  -          JMP DEEK
360  -          JMP DOKE
370  -          JMP SAV

READY.
```

Listing 5. Einfache Parameterübergabe aus Basic-Programmen

sinnvoll, wenn mehrere Leute zusammen an einem größeren Programm arbeiten. Ein Programmierer kann seinen Kollegen bereits feste Einsprungstellen für Routinen, an denen er noch arbeitet oder die noch gar nicht existieren, zur Verfügung stellen.

PRINT AT

Das Programm (Listing 6) ermöglicht eine freie und schnelle Cursorpositionierung zusammen mit einer Druckausgabe. Mit der Definition

```
PR=123*4096 :REM Startadresse
```

kann mit

```
SYSPR,Zeile,Spalte,Printliste
```

alles ausgeben werden, was auch mit PRINT ausgegeben

```
380  -;
390  -;---------------------------------
400  -; PRINT AT
410  -; AUFRUF: SYSPR,ZEILE,SPALTE,PRINTLISTE
420  -;---------------------------------
430  -PRINTAT    JSR CHKKOM     ;1. KOMMA
440  -           JSR XBYTE      ;ZEILE NACH X
450  -           TXA
460  -           PHA            ;AUF STACK ZWISCHENSPEICHERN
470  -           JSR CHKKOM     ;2. KOMMA
480  -           JSR XBYTE      ;SPALTE NACH X
490  -           TXA
500  -           TAY            ;SPALTE NACH Y
510  -           PLA
520  -           TAX            ;ZEILE NACH X
530  -           CLC
540  -           JSR PLOT       ;CURSORPOSITION SETZEN
550  -           JSR CHKKOM     ;3.KOMMA
560  -           JMP PRINT      ;WEITER MIT BASIC-PRINT

READY.
```

Listing 6. PRINT AT-Befehl selbstgemacht

```
570  -;
580  -;---------------------------------
590  -; DEEK (16-BIT-PEEK)
600  -; DER USR VEKTOR ($311/$312)
610  -; MUSS AUF DIESES PROGRAMM ZEIGEN
620  -; AUFRUF: USR(ADRESSE)
630  -;---------------------------------
640  -DEEK       JSR GETADR     ;FAC NACH INTEGER ($14/15)
650  -           LDY #0
660  -           SEI
670  -           LDA ($14),Y    ;LOW-BYTE
680  -           STA $63        ;FAC MANTISSE
690  -           INY
700  -           LDA ($14),Y    ;HIGH-BYTE
710  -           CLI
720  -           STA $62        ;FAC MANTISSE
730  -           LDX #$90       ;FAC EXPONENT
740  -           SEC            ;NICHT INVERTIEREN
750  -           JMP $BC49      ;FAC KOMPLETT MACHEN
```

Listing 7. Eigener DEEK-Befehl

werden kann. Man lasse sich einmal von der Geschwindigkeit des folgenden Programms beeindrucken:

```
10 FOR I=1 TO 24:SYSPR, I,I,"A":NEXT
20 FOR I=1 TO 24:SYSPR,-I,I,"B":NEXT
```

Das Assemberlisting zu PRINT AT bedarf keiner großen Erläuterung. PLOT ist eine Kernel-Funktion, mit der man die Cursorposition auf dem Bildschirm setzen kann. Parameter sind Zeilen- und Spaltennnummern in den Registern X und Y. Das Programm PRINT AT ist eigentlich nicht mehr als eine geschickte Kombination der Routinen PLOT und PRINT.

DEEK (Doppelbyte-PEEK)

Dieses Programm (Listing 7) ist eine Abänderung der PEEK-Routine. DEEK liefert einen 16-Bit-Speicherinhalt an Basic zurück. DEEK wird durch

```
X=USR(Adresse)
```

aufgerufen. Mit Adresse ist die Adresse des Low-Bytes gemeint. Da USR einen Wert zurückgibt, darf es nicht isoliert dastehen, sondern muß als rechte Seite einer Zuweisung oder als Funktionsargument eingesetzt werden. Vor dem ersten Aufruf muß der USR-Vektor auf die Startadresse des Programms gestellt werden:

```
POKE 785,3   :REM LOW-BYTE  $03
POKE 786,192 :REM HIGH-BYTE $C0
```

Im Assemblerlisting steckt eine Besonderheit: Die Zugriffe auf die beiden zu lesenden Bytes (LDA ($14),Y) sind durch ein SEI/CLI-Paar eingerahmt. SEI sperrt die CPU für Interruptanforderungen. Dadurch wird garantiert, daß die beiden Lesezugriffe nicht durch ein Interruptprogramm, welches eines oder beide Bytes ändern könnte, unterbrochen werden können. CLI löst die Interruptsperre wieder.

DOKE (Doppelbyte-POKE)

Um in Basic-Programmen 16-Bit-Größen (zum Beispiel Adressen, Vektoren) in den Speicher zu schreiben, muß man sie vorher erst umständlich in High- und Low-Byte zerlegen, um dann beide Byte POKEn zu könen. Dazu wird meistens die Sequenz:

```
HI=INT(X/256)
LO=X-256*HI
POKE AD,LO
POKE AD+1,HI
```

verwendet.

Wenn man bedenkt, daß jeder Befehl interpretiert werden muß und daß jede Rechenoperation (auch »+1«) in voller Fließkomma-Genauigkeit durchgeführt wird, versteht man, daß dazu viel Rechenzeit nötig ist. Das kleine Maschinenprogramm (Listing 8), das keiner Erläuterung mehr bedarf (FRMNUM, GETADR und CHKKOM sind bekannt) zeigt, wie es einfacher geht:

```
SYSDO,AD,Y
```

```
760  -;
770  -;---------------------------------
780  -; DOKE (16-BIT-POKE)
790  -; AUFRUF: SYSDO,ADRESSE,WERT
800  -;---------------------------------
810  -DOKE       JSR CHKKOM     ;1. KOMMA
820  -          JSR FRMNUM     ;ADRESSE NACH FAC
830  -          JSR GETADR     ;FAC NACH INTEGER ($14/15)
840  -          LDA $14
850  -          STA $9E        ;ADRESSE NACH $9E/9F
860  -          LDA $15
870  -          STA $9F
880  -          JSR CHKKOM     ;2.KOMMA
890  -          JSR FRMNUM     ;WERT NACH FAC
900  -          JSR GETADR     ;FAC NACH INTEGER ($14/15)
910  -          LDY #0
920  -          SEI
930  -          LDA $14        ;WERT LOW-BYTE
940  -          STA ($9E),Y
950  -          INY
960  -          LDA $15        ;WERT HIGH-BYTE
970  -          STA ($9E),Y
980  -          CLI
990  -          RTS

READY.
```

Listing 8. Eigener DEEK-Befehl

(Natürlich muß man DO einmal vorher definieren: DO=12*4096+6). Auch bei DOKE werden die beiden kritischen STA-Befehle durch ein SEI/CLI-Paar untrennbar gemacht. Mit DOKE kann man daher sogar den Interrupt-Vektor ändern. Versucht man dies dagegen mit Hilfe zweier POKEs, kann es passieren, daß ein Interrupt gerade dann auftritt, nachdem das Low-Byte aber noch nicht das High-Byte geändert worden ist. Der Interrupt führt dann auf eine unbestimmte Adresse, was meistens einen Programmabsturz nach sich zieht.

DEEK und DOKE können natürlich auch verschachtelt eingesetz werden. So kann man mit

    SYSDO,A2,USR(A1)

einen 16-Bit-Wert von der Stelle A1 nach A2 kopieren.

Speichern beliebiger Speicherbereiche auf Diskette

Das Programm (Listing 9) realisiert das Gegenstück zum Basic-Befehl:

    LOAD "Name",8,1

```
1000 -;
1010 -;--------------------------------
1020 -; SAV
1030 -; SPEICHERE BELIEBIGEN BEREICH AUF DISK
1040 -; AUFRUF: SYSSAV,DATEINAME,GERAETENUMMER,STARTADRESSE,ENDADRESSE
1050 -;--------------------------------
1060 -SAV       JSR CHKKOM     ;1. KOMMA
1070 -          JSR $E257      ;FILENAMEN HOLEN UND SETZEN
1080 -          JSR CHKKOM     ;2. KOMMA
1090 -          JSR XBYTE      ;GERAETENUMMER NACH X
1100 -          LDY #0         ;SEKUNDAERADRESSE
1110 -          JSR SETLFS     ;FILEPARAMETER SETZEN
1120 -          JSR CHKKOM     ;3. KOMMA
1130 -          JSR FRMNUM     ;STARTADRESSE
1140 -          JSR GETADR     ;NACH $14/15 UND Y/A
1150 -          PHA            ;HIGH-BYTE
1160 -          TYA
1170 -          PHA            ;LOW-BYTE
1180 -          JSR CHKKOM     ;4. KOMMA
1190 -          JSR FRMNUM     ;ENDADRESSE
1200 -          JSR GETADR     ;NACH $14/15 UND Y/A
1210 -          PHA
1220 -          TYA
1230 -          TAX
1240 -          PLA
1250 -          TAY            ;ENDADRESSE LOW IN X, HIGH IN Y
1260 -          PLA            ;STARTADRESSE LOW-BYTE
1270 -          STA $14
1280 -          PLA            ;STARTADRESSE HIGH-BYTE
1290 -          STA $15        ;STARTADRESSE IN $14/15
1300 -          LDA #$14       ;ADRESSE DER STARTADRESSE
1310 -          JSR SAVE
1320 -          BCC SAVENDE    ;KEIN FEHLER
1330 -          JMP $E0F9      ;FEHLERAUSGANG
1340 -SAVENDE   RTS

READY.
```

Listing 9. Speichern von beliebigen Speicherbereichen

Aufgerufen wird es durch:

    SYSSAV,Dateiname,gn,sa,ea

Dabei kann bei »Dateinamen« ein Name oder ein Stringausdruck in Anführungszeichen stehen.
»gn« ist die Gerätenummer (8 oder 9)
»sa« und »ea« sind Start- und Endadresse des abzuspeichernden Bereiches. Zum Programm selbst:

Die Routine bei $E257 beschafft sich den Filenamen aus dem Basic-Text und stellt ihn der später folgenden SAVE-Routine zur Verfügung. Mit SETLFS kann man dem Betriebssystem eine logische Filenummer (im Akku), eine Gerätenummer (in X) und eine Sekundäradresse (in Y) bekanntgeben. Die Parametrisierung der Kernel-SAVE-Routine ist etwas komplizierter:

X Endadresse Low-Byte
Y Endadresse High-Byte
A Zeiger auf das untere Byte eines Zero-Page-Bytepaares, welches die Startadresse enthält.

Die SAVE-Routione kehrt mit gesetztem Carry-Flag zurück, falls beim Speichern ein Fehler aufgetreten ist. Das Programm bei $E0F9 sorgt dann für eine ordentliche Fehlermeldung.

```
100  -;--------------------------------
110  -; MULTIPLIKATION  8 MAL 8 BIT
120  -;
130  -;  MD MULTIPLIKAND (BLEIBT ERHALTEN)
140  -;  MR MULTIPLIKATOR (WIRD UEBERSCHRIEBEN)
150  -;
160  -; DAS 16-BIT-PRODUKT STEHT IN:
170  -;  MR (HIGH-BYTE) UND
180  -;  A  (LOW-BYTE)
190  -;--------------------------------
200  -          .BA $C000
210  -          .EQ MD=$FD
220  -          .EQ MR=$FE
230  -MUL       LDA #0         ;VORBESETZUNG DES PRODUKTS
240  -          LDX #8         ;ZAEHLER (8 DURCHLAEUFE)
250  -MULLOOP   ASL            ;PRODUKT IN A UEBER
260  -          ROL MR         ;MR NACH LINKS SCHIEBEN
270  -          BCC MULNEXT    ;HOECHSTES BIT IN MR=0
280  -          CLC            ;FALLS HOECHSTES BIT IN MR=1,
290  -          ADC MD         ;MD ZUM TEILPRODUKT ADDIEREN
300  -          BCC MULNEXT    ;KEIN UEBERTRAG
310  -          INC MR         ;UEBERTRAG NACH MR BERUECKSICHTIGEN
320  -MULNEXT   DEX
330  -          BNE MULLOOP    ;WEITER, FALLS ZAEHLER NOCH NICHT 0
340  -          RTS

READY.
```

Listing 10. Mulitplikation 8 mal 8 Bit

Multiplikation

Das Programm (Listing 10) multipliziert zwei Byte-Werte miteinander und liefert ein 16-Bit-Produkt. Es ist aber trotz seiner Kürze nicht ganz einfach zu verstehen. Die beiden zu multiplizierenden Faktoren seien mit

    Multiplikator MR und Mulitplikand MD

bezeichnet. Für das Resultat ist es natürlich gleichgültig, welcher Faktor als MR und welcher als MD an das Programm übergeben wird. MR kann man sich, wie jede binäre Größe, folgendermaßen vorstellen:

MR=MR(7)*128+MR(6)*64+...+MR(1)*2+MR(0)*1

Dabei bezeichnet zum Beispiel MR(6) das Bit Nummer 6 von MR in der üblichen Zählweise von 0 bis 7 und von rechts nach links. Das Produkt MR*MD kann man nun so berechnen:

Addiere folgende Teilprodukte:

    MD*128, falls MR(7)=1, sonst 0
    MD*64, falls MR(6)=1, sonst 0
    MD*2,   falls MR(1)=1, sonst 0
    MD,   falls MR(0)=1, sonst 0

Die Teilprodukte erhält man einfach durch Linksverschieben von MD:

    MD*128 durch 7-maligen Links-Shift
    MD*64 durch 6-maligen Links-Shift

Zu addieren ist nur dann etwas, wenn das entsprechende Bit in MR=1 ist. Wenn man MR mit dem ROL-Befehl achtmal

nach links schiebt, so durchwandern alle 8 Bit nacheinander das Carry-Flag und letzteres kann leicht abgefragt werden. Der Trick des Programms besteht nun darin, daß das Berechnen von Teilproduktsummen mit dem Linksschieben von MR kombiniert wird. Zunächst wird die Zwischensumme in A mit 0 vorbesetzt. MR wird nach links geschoben. Das höchstwertige Bit von MR steht jetzt im Carry-Flag. Ist es 1, so wird MD zum Akku addiert. Eigentlich müßte jetzt der Akku um sieben Positionen nach links geschoben werden, da zum ersten Teilprodukt der Faktor 128 gehört. Diese Verschiebung ergibt sich aber automatisch im Verlauf der nächsten sieben Schleifendurchläufe.

Sowie MR nach links geschoben wird, werden rechts in MR Bits frei, die dann von den von rechts kommenden höherwertigen Bits der Zwischensumme belegt werden. Nach 8 Schleifendurchläufen ist schließlich MR nach links verdrängt worden. An seiner Stelle steht nun das High-Byte des Produkts. Das Low-Byte des Produkts steht im Akku, während MD unverändert geblieben ist.

```
100  -;--------------------------------
110  -; SCHIEBEREGISTER-FOLGEN
120  -; ALS PSEUDO-ZUFALLSZAHLEN
130  -;--------------------------------
140  -          .BA $C000
150  -          .EQ SR=$FD     ;SCHIEBEREGISTER (2 BYTE)
160  -          .EQ ZAEHLER=$FB;(2 BYTE)
170  -          .EQ DELAY=$FA  ;PAUSENLAENGE
180  -;
190  -          JMP MAIN       ;ZUM HAUPTPROGRAMM
200  -;
210  -; NAECHSTE ZUFALLSZAHL
220  -;  (1) CARRY := SR(6) EOR SR(9)
230  -;  (2) SR NACH LINKS SCHIEBEN
240  -;  (3) SR(0) := CARRY
250  -;
260  -SHIFT     LDA SR+1
270  -          AND #2         ;SR(9) ISOLIEREN
280  -          ASL
290  -          ASL
300  -          ASL            ;IN BITPOSITION 6
310  -          ASL            ;BRINGEN
320  -          ASL
330  -          EOR SR         ;LIEFERT SR(6) EOR SR(9)
340  -          ASL
350  -          ASL            ;RESULTAT INS CARRY-FLAG (SCHRITT (1))
360  -          ROL SR         ;SCHRITT (2) UND (3)
370  -          ROL SR+1
380  -          RTS
390  -;
400  -; VERZOEGERUNGSSCHLEIFE
410  -;
420  -PAUSE     LDX DELAY
430  -PAUSE1    DEX
440  -          BNE PAUSE1
450  -          RTS
460  -;
470  -; HAUPTPROGRAMM
480  -; SCHIEBEREGISTER-FOLGE DER LAENGE 1023 ERZEUGEN
490  -; BILDSCHIRMZEICHEN IN DER REIHENFOLGE DIESER ZUFALLSZAHLEN
500  -; INVERTIEREN (D.H. BIT 7 INVERTIEREN)
510  -;
520  -; SR MIT ZUFAELLIGEM STARTWERT VORBESETZEN
530  -MAIN      LDA $DC04      ;CIA#1 TIMER A, LOW-BYTE
540  -          ORA #1         ;DARF NICHT 0 SEIN
550  -          STA SR
560  -          LDA #$FF
570  -          STA ZAEHLER
580  -          LDA #$03
590  -          STA ZAEHLER+1  ;ZAEHLER=$3FF=1023
600  -LOOP      JSR SHIFT      ;NAECHSTE ZUFALLSZAHL
610  -          LDA SR+1
620  -          PHA            ;MERKEN
630  -          AND #3         ;HIGH-BYTE AUF 2 BIT BEGRENZEN
640  -          ORA #4         ;SR=SR+$400
650  -          STA SR+1
660  -          LDY #0
670  -          LDA (SR),Y     ;ZEICHEN VOM BILDSCHIRM
680  -          EOR #$80       ;BIT 7 INVERTIEREN
690  -          STA (SR),Y     ;ZURUECK ZUM BILDSCHIRM
700  -          PLA            ;SR+1
710  -          STA SR+1       ;WIEDERHERSTELLEN
720  -          JSR PAUSE
730  -          DEC ZAEHLER
740  -          BNE LOOP
750  -          DEC ZAEHLER+1
760  -          BPL LOOP
770  -          LDA $400       ;ERSTES BILDSCHIRMZEICHEN
780  -          EOR #$80       ;INVERTIEREN
790  -          STA $400
800  -          RTS
```

Listing 11. Schieberegister-Folgen als Pseudo-Zufallszahlen

Erwähnenswert sind hier noch die Befehle ASL und ROL: Beide schieben nach links und bei beiden wird Bit 7 ins Carry-Flag geschoben. Der Unterschied:

ASL besetzt Bit 0 mit 0

ROL besetzt Bit 0 mit dem alten Inhalt des Carry-Flags.

Mit ASL (ohne Adreßteil) wird also der Akku arithmetisch verdoppelt, während mit ROL MR zusätzlich der Übertrag aus dieser Verdoppelung in Bit 0 von MR gelangt.

Schieberegister-Folgen als Pseudo-Zufallszahlen

Das Programm (Listing 11) zeigt eine interessante Anwendung von Schieberegistern. Wenn man ein Schieberegister (SR) an den »richtigen« Bitpositionen »anzapft« und das Exklusiv-Oder-Produkt dieser Bits an den SR-Eingang zurückführt, erhält man eine Folge von Bits, die vollkommen zufällig zu sein scheint. Die Folgen sind zwar periodisch, sie wiederholen sich also nach einer gewissen Zeit, die Periodenlänge kann aber beliebig lang gemacht werden. Macht man eine so erzeugte 0-1-Folge mit einem Lautsprecher hörbar, so klingt diese wie weißes Rauschen.

Die folgende Tabelle enthält geeignete Anzapfstellen für Schieberegister unterschiedlicher Länge.

Registerlänge	Rückkopplung	Periodenlänge
2	0+1	3
3	1+2	7
4	2+3	15
5	2+4	31
6	4+5	63
7	5+6	127
8	1+2+3+7	255
9	4+8	511
10	6+9	1023
11	8+10	2047
12	1+9+10+11	4095
13	0+10+11+12	8191
14	1+11+12+13	16383
15	13+14	32767
16	10+12+13+15	65535

»+« steht hi[illegible] »EOR«

Die angegebenen Periodenlängen sind die bei der jeweiligen Registerlänge maximal möglichen. Die Schieberegisterfolgen haben die angenehme Eigenschaft, daß die Registerwerte alle Zahlen von 1 bis zur Periodenlänge in quasizufälliger Reihenfolge durchlaufen. Man darf ein solches Schieberegister allerdings nicht mit lauter Nullen vorbesetzen, da es dann seinen Zustand nicht mehr ändert (0 EOR 0 =0).

Der Kern des folgenden Programms ist die kleine Routine SHIFT. Die beiden Zero-Page-Speicherstellen SR und SR+1 bilden ein 16-Bit-Schieberegister. Rückgekoppelt wird es an den Positionen 6 und 9. Es werden quasi nur 10 Bit von den 16 vorhandenen ausgenutzt. SHIFT erzeugt bei wiederholtem Aufruf eine Folge mit der Periode 1023.

Das Hauptprogramm wendet nun diese Folge in grafisch reizvoller Weise an. Zunächst wird das Low-Byte des Schieberegisters mit einem zufälligen Wert (ungleich 0) vorbesetzt. Dieser Wert stammt aus dem ständig laufenden Timer A in CIA Nummer 1. SHIFT wird nun 1023 mal aufgerufen und erzeugt dadurch alle Zahlen von 1 bis 1023 in quasizufälliger Reihenfolge. Diese Zahlen werden als Adressen relativ zum Bildschirm-RAM verwendet. Bei den adressierten Bytes wird jeweils Bit 7 invertiert, was eine Reversdarstellung der Bildschirmzeichen bewirkt. Das Programm MAIN bewirkt also nichts anderes als eine Invertierung des gesamten Textbildschirms. Da dies aber in zufälliger Abfolge geschieht, ist der Effekt sehr auffallend. Über die Variable DELAY ($FA)=250) kann man das Tempo der Invertierung beeinflussen. Das anschließende Basic-Programm (Listing 12) erzeugt einen Flimmereffekt, indem es eine einfache Zufallsgrafik mit dem Programm MAIN invertiert. (Thomas Krätzig/aw)

```
10 POKE 250,1
20 PRINT CHR$(147)
30 FOR I=1024 TO 2024
40 POKE I,127+INT(RND(0)*2)*128
50 NEXT I
60 SYS 12*4096
70 GET A$:IF A$="" THEN 60
READY.
```

Listing 12. Basic-Hilfsprogramm zu Listing 11.

Superhirn, einmal andersherum

Lassen Sie den C 64 Ihre Kombination herausfinden. Nach einem perfekten Algorithmus kann der C 64 Ihre Zahlenfolge in maximal sechs Versuchen berechnen.

Beim Spiel Superhirn geht es darum, einen von einem Mitspieler ausgedachten Farbcode zu erraten. Man probiert verschiedene Farbcodes aus; der Mitspieler gibt nach einer versuchten Kombination Information über deren Richtigkeit.

Es gibt schon einige Superhirn-Programme, die eine Kombination wählen, die dann der menschliche Mitspieler erraten muß.

Superhirn II geht den anderen Weg: Der Mensch denkt sich eine Kombination aus, die dann der Computer erraten muß! Superhirn II findet jede Kombination garantiert (siehe Kasten).

Die Farben werden durch Zahlen symbolisiert. Es gibt sechs »Farbzahlen«, nämlich die Ziffern 1, 2, 3, 4, 5 und 6. Für die Farben gibt es vier Positionen

Bedienungsanleitung:

Geben Sie Superhirn II bitte mit dem MSE ein (Listing 1) und starten Sie es mit »SYS 13312«.

Sie antworten, indem Sie zuerst die Anzahl der richtigen Farbziffern an der richtigen Position eingeben und dann mit »RETURN« abschließen. Dann geben Sie die Anzahl der richtigen Farbziffern an der falschen Position ein und bestätigen wiederum mit »RETURN«.

Superhirn II gibt dann seinen neuen Lösungsvorschlag aus; Sie antworten wieder wie oben. Das Programm ist beendet, wenn

- das Programm die richtige Kombination gefunden hat (es meldet sich mit »OK«)
- oder wenn Sie eine unmögliche Antwort gegeben haben (das Programm meldet sich mit »ERROR«).

Beispiel:

Nehmen wir die Kombination »3436«. Nach Programm-Start beginnt Superhirn II wie üblich mit seiner Startkombination »1234«.

Der Computer hat die »3« richtig als Zahl und Position; die »4« stimmt nur als Zahl.

Also wird eingegeben:

1234 – der Vorschlag des Computers.
1 < RETURN > – für die richtige »3«.
1 < RETURN > – für die »4« an falscher Position.

Jetzt sucht der Computer einen neuen Lösungsvorschlag. Verfolgen wir die Eingaben bis zum Ende ...

2211 – der 2. Versuch des Computers.
0 < RETURN > – keine richtige Position/richtige Ziffer.
0 < RETURN > – keine falsche Position/richtige Ziffer.
4333 – der 3. Versuch des Computers.
1 < RETURN > – für die richtige »3«.
2 < RETURN > – für die »3« und »4«.
3344 – der 4. Versuch des Computers.
1 < RETURN > – diesmal stimmt die erste »3«.
2 < RETURN > – dafür sind »3« und »4« vertauscht.
3435 – der 5. Versuch des Computers.
3 < RETURN > – bis auf die fehlende »6« richtig.
0 < RETURN > – die »5« ist falsch.
3436 – der 6. Versuch des Computers.
4 < RETURN > – die Kombination ist gefunden.
OK – Sie befinden sich wieder im Basic.

Das Programm kann mit »SYS 13312« wieder gestartet werden.

Hinweise:

Die Eingabe unlogischer Daten führt zur Ausgabe von »ERROR«. Sie haben dann entweder eine offensichtlich falsche Eingabe gemacht (zum Beispiel bei der Eingabe einer »6« für die Anzahl der richtigen Ziffern – es kann ja nur höchstens »4« als korrekte Eingabe vorkommen!) oder eine logisch falsche Eingabe.

Was ist eine logisch falsche Eingabe? Nehmen wir unser Beispiel von oben:

Geben Sie nach dem 6. Versuch des Computers nicht die (richtige!) »4« ein, sondern »0« und nochmal »0«, antwortet Ihnen der Computer mit »ERROR«. Warum?

Nun gibt es einfach keine Ziffernkombination mehr, die allen Ihren Eingaben gerecht werden kann. Folglich müssen Sie einen Eingabe-Fehler gemacht haben.

Geben Sie in unserem Beispiel nach dem 3. Versuch des Computers nicht »1« und »2« (was nach Wahl unserer Ziffernkombination »3436« die einzig korrekte Eingabe ist), son-

```
programm : superhirn ii      3400 360c
---------------------------------------

3400 : a9 93 20 d2 ff a9 01 8d   42
3408 : 3e 03 8d 52 03 8d 5c 03   8a
3410 : 8d 66 03 8d 70 03 8d 53   3f
3418 : 03 0a 8d 5d 03 0a 8d 67   b5
3420 : 03 0a 8d 71 03 a9 df a0   f8
3428 : 35 20 1e ab 20 ac 35 c0   28
3430 : ff d0 01 60 4c 3b 35 ad   b3
3438 : 52 03 29 20 d0 06 0e 52   74
3440 : 03 4c 3b 35 a9 01 8d 52   5c
3448 : 03 ad 5c 03 29 20 d0 06   7c
3450 : 0e 5c 03 4c 3b 35 a9 01   dd
3458 : 8d 5c 03 ad 66 03 29 20   ed
3460 : d0 06 0e 66 03 4c 3b 35   6e
3468 : a9 01 8d 66 03 ad 70 03   27
3470 : 29 20 d0 06 0e 70 03 4c   a7
3478 : 3b 35 a9 6b a0 a3 20 1e   09
3480 : ab 60 a9 00 9d 84 03 ad   2b
3488 : 52 03 3d 52 03 f0 03 fe   b7
3490 : 84 03 ad 5c 03 3d 5c 03   1e
3498 : f0 03 fe 84 03 ad 66 03   97
34a0 : 3d 66 03 f0 03 fe 84 03   2f
34a8 : ad 70 03 3d 70 03 f0 03   df
34b0 : fe 84 03 60 ad 52 03 8d   52
34b8 : 3f 03 ad 5c 03 8d 40 03   13
34c0 : ad 66 03 8d 41 03 ad 70   d7
34c8 : 03 8d 42 03 bd 52 03 8d   18
34d0 : 43 03 bd 5c 03 8d 44 03   43
34d8 : bd 66 03 8d 45 03 bd 70   7f
34e0 : 03 8d 46 03 a9 00 8d 49   ff
34e8 : 03 a0 06 a9 00 8d 47 03   82
34f0 : 8d 48 03 8a 48 a9 03 aa   e7
34f8 : 5e 3f 03 90 03 ee 47 03   94
3500 : 5e 43 03 90 03 ee 48 03   a2
3508 : ca 10 ed 68 aa ad 47 03   9e
3510 : 38 ed 48 03 30 0d ad 49   66
3518 : 03 18 6d 48 03 8d 49 03   54
3520 : 4c 2d 35 ad 49 03 18 6d   ee
3528 : 47 03 8d 49 03 88 d0 bb   ad
3530 : ad 49 03 38 fd 84 03 9d   95
3538 : 98 03 60 ae 3e 03 20 82   c1
3540 : 34 bd 84 03 dd 7a 03 f0   74
3548 : 03 4c 37 34 20 b4 34 bd   ba
3550 : 98 03 dd 8e 03 f0 03 4c   0f
3558 : 37 34 ca d0 e1 ae 3e 03   08
3560 : e8 ad 52 03 9d 52 03 8d   a7
3568 : 43 03 ad 5c 03 9d 5c 03   b8
3570 : 8d 44 03 ad 66 03 9d 66   58
3578 : 03 8d 45 03 ad 70 03 9d   99
3580 : 70 03 8d 46 03 a0 00 a9   27
3588 : 00 aa 18 c8 b9 42 03 4a   4b
3590 : e8 90 fc 8a 18 69 30 20   1f
3598 : d2 ff c0 04 d0 e9 ee 3e   af
35a0 : 03 20 ac 35 c0 ff d0 01   d6
35a8 : 60 4c 37 34 a9 0d 20 d2   ac
35b0 : ff 20 cf ff ae 3e 03 38   0c
35b8 : e9 30 c9 04 d0 0a a9 64   79
35c0 : a0 a3 20 1e ab a0 ff 60   7e
35c8 : 9d 7a 03 a9 0d 20 d2 ff   b5
35d0 : 20 cf ff 38 e9 30 9d 8e   92
35d8 : 03 a9 0d 20 d2 ff 60 31   08
35e0 : 32 33 34 00 a9 00 85 6a   3e
35e8 : 85 6c a9 04 85 6b a9 34   51
35f0 : 85 6d a0 00 b1 6a 91 6c   e1
35f8 : e8 d0 f9 a5 6b c9 07 f0   5e
3600 : 07 e6 6b e6 6d 18 90 ea   e2
3608 : 4c 00 34 ff aa aa aa aa   61
```

Listing 1. Das MSE-Listing zu »Superhirn II«. Bitte beachten Sie die Eingabehinweise auf Seite 110

dern »2« und »2«, kann Fehleingabe natürlich nicht mit einem »ERROR« beantwortet werden, weil nämlich jetzt noch Ziffernkombinationen vorhanden sind, die nicht im Widerspruch zu Ihren bis dahin gemachten Eingaben stehen.

Zum Programm:

Den Quelltext des Programms entnehmen Sie Listing 2. Die Eingabe-Routine liest nur das erste Zeichen jeder Zeile – es macht also keinen Unterschied, ob man als Antwort »1« oder »123456« eingibt, da die restlichen Zeichen nicht berücksichtigt werden.

Das Programm ist leicht auf zum Beispiel 10 Positionen erweiterbar – man muß jedoch folgendes beachten: Die Abstände zwischen den Registern POS1, ..., CWEISSE müssen vergrößert werden, da möglicherweise neun Versuche nicht zum Ziel führen.

Auch die Anzahl der möglichen Ziffern kann leicht von sechs auf acht erweitert werden. Möchte man mehr als acht Ziffern, muß man mit 16-Bit-Zahlen arbeiten, was größere Umschreibarbeiten zur Folge hat.

Kombinations-Codierung:

Es bleibt noch die Frage, in welcher Art und Weise die Kombinationen gespeichert werden. Nehmen wir dazu doch wieder unsere Kombination aus dem Beispiel, also »3436«.

In POS1, POS2, POS3 und POS4 steht immer die aktuelle Kombination, die der Computer gerade als Lösungsversuch ausgegeben hat.

Ist das Problem gerade bei der Lösung angelangt, dann steht nicht etwa in POS1 die »3«, in POS2 die »4«, in POS3 wieder die »3« und in POS4 die »6« – sondern die Kombination ist folgendermaßen festgehalten:

POS1 – 2 hoch (3–1)
POS2 – 2 hoch (4–1)
POS3 – 2 hoch (3–1)
POS4 – 2 hoch (6–1)

Durch diese Darstellung ist es später leichter möglich, die Anzahl der Schwarzen (= richtige Ziffer/richtige Position) und die Anzahl der Weißen (= richtige Ziffer/falsche Position) zu bestimmen.

Der Lösungsversuch des Computers in der x-ten Runde steht genauso codiert in POS1+X, POS2+X, POS3+X und POS4+X.

(A. Reiser/H. Bauschke/og)

Das Programm gibt den ersten Lösungsvorschlag aus. Nach der Eingabe des menschlichen Mitspielers wird eine Ziffernkombination gesucht, die logisch richtig ist in bezug auf alle vorher gemachten Eingaben.

Dabei werden alle Ziffernkombinationen von »1111« bis »6666« getestet.

Ist die aktuelle Ziffernkombination logisch richtig, gibt sie der Computer aus.

Das Programm beginnt bei der nächsten Suche nach der Lösung bei der zuletzt ausgegebenen Ziffernkombination; es muß also nicht wieder bei »1111« anfangen.

Hat der menschliche Gegner nicht mit »4« bei richtiger Position/richtige Farbe geantwortet und ist das Programm bei »6666« angelangt, kann es sich nur um eine falsche Eingabe handeln – es wird »ERROR« ausgegeben.

Wie beurteilt das Programm nun, ob es in seiner Schleife von »1111« bis »6666« gerade bei einer logisch richtigen oder logisch falschen Ziffernkombination ist?

Dazu benutzt der Algorithmus folgenden Trick: Es wird angenommen, daß die zu prüfende Ziffernkombination die richtige ist. Dann werden alle bis dahin getätigten Eingaben mit dieser Kombination verglichen. Es wird also nach der Anzahl der richtigen Ziffer/richtige Position und nach der Anzahl richtige Ziffer/falsche Position gesucht. Das Ergebnis wird festgehalten und mit den Eingaben, die Sie gemacht haben, verglichen.

Stimmen diese Werte sämtlich überein, kann es sich bei der zu prüfenden Kombination um die Lösung handeln – sie muß es aber nicht sein!

Trifft nun das Programm auf eine solche Lösungsmöglichkeit, gibt es diese auch aus.

Ist die Prüfung negativ ausgefallen, das heißt, es gab mindestens eine Abweichung von der tatsächlichen Eingabe des menschlichen Mitspielers, wird in der großen Schleife weitergesucht – so lange, bis die Kombination gefunden ist.

So findet das Programm die Kombination

```
105:  3400                    .TIT "SUPERHIRN II"
110:  3400                    .OPT OO,P1
120:  3400                    *=   $3400
121:                    ;
122:                    ; VARIABLEN-DEKLARATION
123:                    ;
125:  006A              ALT      =    $6A     ; ZEROPAGE ADRESSEN FUER
126:  006C              NEU      =    $6C     ; VERSCHIEBE-ROUTINE
130:  FFD2              CHROUT   =    $FFD2
140:  FFCF              CHRIN    =    $FFCF
150:  AB1E              STROUT   =    $AB1E
160:  0352              POS1     =    850     ; SPEICHER FUER
170:  035C              POS2     =    860     ; KOMBINATIONEN
180:  0366              POS3     =    870
190:  0370              POS4     =    880
200:  037A              SCHWARZE =    890     ; EINGABEN MITSPIELER
210:  0384              CSCHWARZE=    900     ; ERRECHNETE EINGABEN
220:  038E              WEISSE   =    910     ; ( SIEHE ALGORITHMUS )
230:  0398              CWEISSE  =    920     ;
240:  033E              RUNDE    =    830
245:                    ; DIVERSE HILFSREGISTER
250:  033F              CPOS1    =    831
260:  0340              CPOS2    =    832
270:  0341              CPOS3    =    833
280:  0342              CPOS4    =    834
290:  0343              CV1      =    835
300:  0344              CV2      =    836
310:  0345              CV3      =    837
320:  0346              CV4      =    838
330:  0347              H1       =    839
340:  0348              H2       =    840
360:  0349              MIN      =    841
370:                    ;
372:                    ; PROGRAMM-START
374:                    ;
380:  3400 A9 93                  LDA  #147
390:  3402 20 D2 FF               JSR  CHROUT   ; BILDSCHIRM LOESCHEN
470:  3405 A9 01                  LDA  #1
475:  3407 8D 3E 03               STA  RUNDE    ; RUNDE INITIALISIERT
480:  340A 8D 52 03               STA  POS1
480:  340D 8D 5C 03               STA  POS2
480:  3410 8D 66 03               STA  POS3
480:  3413 8D 70 03               STA  POS4     ; SCHLEIFE INITIALISIERT
                                                    AUF '1111'

485:                    ;
486:                    ; 1. VERSUCH '1234' ABSPEICHERN UND AUSGEBEN
487:                    ;
490:  3416 8D 53 03               STA  POS1+1
500:  3419 0A                     ASL  A
500:  341A 8D 5D 03               STA  POS2+1
510:  341D 0A                     ASL  A
510:  341E 8D 67 03               STA  POS3+1
520:  3421 0A                     ASL  A
520:  3422 8D 71 03               STA  POS4+1
540:  3425 A9 DF                  LDA  #<TEXT
550:  3427 A0 35                  LDY  #>TEXT
560:  3429 20 1E AB               JSR  STROUT
570:  342C 20 AC 35               JSR  ANTWORT
572:  342F C0 FF                  CPY  #$FF     ; KOMBINATION
                                                  GEFUNDEN JA/NEIN

574:  3431 D0 01                  BNE  LO       ; NEIN - ALSO SUCHEN
576:  3433 60                     RTS           ; JA - ZURUECK INS BASIC
580:  3434 4C 3B 35 LO            JMP  LILOOP
600:                    ;
610:                    ; GROSSE SCHLEIFE - '1111' BIS '6666'
620:                    ;
640:  3437 AD 52 03 GRLOOP        LDA  POS1
650:  343A 29 20                  AND  #%00100000
660:  343C D0 06                  BNE  MASSN1
670:  343E 0E 52 03               ASL  POS1
680:  3441 4C 3B 35               JMP  LILOOP
690:  3444 A9 01    MASSN1        LDA  #1
700:  3446 8D 52 03               STA  POS1
710:  3449 AD 5C 03 TEST2         LDA  POS2
720:  344C 29 20                  AND  #%00100000
730:  344E D0 06                  BNE  MASSN2
740:  3450 0E 5C 03               ASL  POS2
750:  3453 4C 3B 35               JMP  LILOOP
760:  3456 A9 01    MASSN2        LDA  #1
770:  3458 8D 5C 03               STA  POS2
780:  345B AD 66 03 TEST3         LDA  POS3
```

Listing 2. Der Quell-Code des »Superhirn II«

```
790:   345E 29 20              AND  #%00100000
800:   3460 D0 06              BNE  MASSN3
810:   3462 0E 66 03           ASL  POS3
820:   3465 4C 3B 35           JMP  LILOOP
830:   3468 A9 01     MASSN3   LDA  #1
840:   346A 8D 66 03           STA  POS3
850:   346D AD 70 03  TEST4    LDA  POS4
860:   3470 29 20              AND  #%00100000
870:   3472 D0 06              BNE  MASSN4
880:   3474 0E 70 03           ASL  POS4
890:   3477 4C 3B 35           JMP  LILOOP
900:   347A A9 6B     MASSN4   LDA  #$6B
900:   347C A0 A3              LDY  #$A3    ; KOMBINATION NICHT
                                              GEFUNDEN
910:   347E 20 1E AB           JSR  STROUT  ; 'ERROR' AUSGEBEN
930:   3481 60                 RTS          ; UND ZURUECK INS BASIC.
932:                  ;
933:                  ; UNTERPROGRAMM ANZAHL SCHWARZE ERMITTELN
934:                  ;
940:   3482 A9 00     BLACKS   LDA  #0
950:   3484 9D 84 03           STA  CSCHWARZE,X
960:   3487 AD 52 03  POS1V    LDA  POS1
970:   348A 3D 52 03           AND  POS1,X
980:   348D F0 03              BEQ  POS2V
990:   348F FE 84 03           INC  CSCHWARZE,X
1000:  3492 AD 5C 03  POS2V    LDA  POS2
1010:  3495 3D 5C 03           AND  POS2,X
1020:  3498 F0 03              BEQ  POS3V
1030:  349A FE 84 03           INC  CSCHWARZE,X
1040:  349D AD 66 03  POS3V    LDA  POS3
1050:  34A0 3D 66 03           AND  POS3,X
1060:  34A3 F0 03              BEQ  POS4V
1070:  34A5 FE 84 03           INC  CSCHWARZE,X
1080:  34A8 AD 70 03  POS4V    LDA  POS4
1090:  34AB 3D 70 03           AND  POS4,X
1100:  34AE F0 03              BEQ  BFIN
1110:  34B0 FE 84 03           INC  CSCHWARZE,X
1120:  34B3 60        BFIN     RTS
1122:                 ;
1124:                 ; UNTERPROGRAMM ANZAHL WEISSE ERMITTELN
1126:                 ;
1130:  34B4 AD 52 03  WHITES   LDA  POS1
1140:  34B7 8D 3F 03           STA  CPOS1
1150:  34BA AD 5C 03           LDA  POS2
1160:  34BD 8D 40 03           STA  CPOS2
1170:  34C0 AD 66 03           LDA  POS3
1180:  34C3 8D 41 03           STA  CPOS3
1190:  34C6 AD 70 03           LDA  POS4
1200:  34C9 8D 42 03           STA  CPOS4
1210:  34CC BD 52 03           LDA  POS1,X
1220:  34CF 8D 43 03           STA  CV1
1230:  34D2 BD 5C 03           LDA  POS2,X
1240:  34D5 8D 44 03           STA  CV2
1250:  34D8 BD 66 03           LDA  POS3,X
1260:  34DB 8D 45 03           STA  CV3
1270:  34DE BD 70 03           LDA  POS4,X
1280:  34E1 8D 46 03           STA  CV4
1290:  34E4 A9 00              LDA  #0
1300:  34E6 8D 49 03           STA  MIN
1310:  34E9 A0 06              LDY  #6
1330:  34EB A9 00     LOS      LDA  #0
1340:  34ED 8D 47 03           STA  H1
1350:  34F0 8D 48 03           STA  H2
1400:  34F3 8A                 TXA
1400:  34F4 48                 PHA
1400:  34F5 A9 03              LDA  #3
1400:  34F7 AA                 TAX
1410:  34F8 5E 3F 03  L1       LSR  CPOS1,X
1420:  34FB 90 03              BCC  L2
1430:  34FD EE 47 03           INC  H1
1440:  3500 5E 43 03  L2       LSR  CV1,X
1450:  3503 90 03              BCC  L3
1460:  3505 EE 48 03           INC  H2
1470:  3508 CA        L3       DEX
1480:  3509 10 ED              BPL  L1
1490:  350B 68                 PLA
1490:  350C AA                 TAX
1600:  350D AD 47 03  MINFIND  LDA  H1
1610:  3510 38                 SEC
1620:  3511 ED 48 03           SBC  H2
1630:  3514 30 0D              BMI  H2GROSS
1640:  3516 AD 49 03  H1GROSS  LDA  MIN
1650:  3519 18                 CLC
1660:  351A 6D 48 03           ADC  H2
1670:  351D 8D 49 03           STA  MIN
1680:  3520 4C 2D 35           JMP  YKLEINER
1690:  3523 AD 49 03  H2GROSS  LDA  MIN
1700:  3526 18                 CLC
1710:  3527 6D 47 03           ADC  H1
1720:  352A 8D 49 03           STA  MIN
1730:  352D 88        YKLEINER DEY
1740:  352E D0 BB              BNE  LOS
1750:  3530 AD 49 03           LDA  MIN
1760:  3533 38                 SEC
1770:  3534 FD 84 03           SBC  CSCHWARZE,X
1780:  3537 9D 98 03           STA  CWEISSE,X

1790:  353A 60        WFIN     RTS
1792:                 ;
1794:                 ; KLEINE SCHLEIFE - KOMBINATION
                        LOGISCH-RICHTIG (J/N)

1796:                 ;
1800:  353B AE 3E 03  LILOOP   LDX  RUNDE
1810:  353E 20 82 34  LISTART  JSR  BLACKS
1820:  3541 BD 84 03           LDA  CSCHWARZE,X
1830:  3544 DD 7A 03           CMP  SCHWARZE,X
1840:  3547 F0 03              BEQ  L4
1845:  3549 4C 37 34           JMP  GRLOOP
1850:  354C 20 B4 34  L4       JSR  WHITES
1860:  354F BD 98 03           LDA  CWEISSE,X
1870:  3552 DD 8E 03           CMP  WEISSE,X
1880:  3555 F0 03              BEQ  L5
1885:  3557 4C 37 34           JMP  GRLOOP
1890:  355A CA        L5       DEX
1900:  355B D0 E1              BNE  LISTART
1902:                 ;
1904:                 ; LOGISCH-RICHTIGE KOMBINATION AUSGEBEN
1906:                 ;
1910:  355D AE 3E 03           LDX  RUNDE
1920:  3560 E8                 INX
1930:  3561 AD 52 03           LDA  POS1
1940:  3564 9D 52 03           STA  POS1,X
1940:  3567 8D 43 03           STA  CV1
1950:  356A AD 5C 03           LDA  POS2
1960:  356D 9D 5C 03           STA  POS2,X
1960:  3570 8D 44 03           STA  CV2
1970:  3573 AD 66 03           LDA  POS3
1980:  3576 9D 66 03           STA  POS3,X
1980:  3579 8D 45 03           STA  CV3
1990:  357C AD 70 03           LDA  POS4
2000:  357F 9D 70 03           STA  POS4,X
2000:  3582 8D 46 03           STA  CV4
2080:  3585 A0 00              LDY  #0
2090:  3587 A9 00     L6       LDA  #0
2090:  3589 AA                 TAX
2095:  358A 18                 CLC
2095:  358B C8                 INY
2100:  358C B9 42 03           LDA  CV1-1,Y
2110:  358F 4A        L7       LSR  A
2120:  3590 E8                 INX
2130:  3591 90 FC              BCC  L7
2140:  3593 8A                 TXA
2140:  3594 18                 CLC
2140:  3595 69 30              ADC  #$30
2150:  3597 20 D2 FF           JSR  CHROUT
2160:  359A C0 04              CPY  #4
2170:  359C D0 E9              BNE  L6
2190:  359E EE 3E 03           INC  RUNDE
2200:  35A1 20 AC 35           JSR  ANTWORT
2202:  35A4 C0 FF              CPY  #$FF
2204:  35A6 D0 01              BNE  L8
2206:  35A8 60                 RTS
2210:  35A9 4C 37 34  L8       JMP  GRLOOP
2212:                 ;
2214:                 ; UNTERPROGRAMM ANTWORT HOLEN
2216:                 ;
2220:  35AC A9 0D     ANTWORT  LDA  #$0D
2220:  35AE 20 D2 FF           JSR  CHROUT
2230:  35B1 20 CF FF           JSR  CHRIN
2240:  35B4 AE 3E 03           LDX  RUNDE
2250:  35B7 38                 SEC
2250:  35B8 E9 30              SBC  #$30
2252:  35BA C9 04              CMP  #4
2254:  35BC D0 0A              BNE  L9
2256:  35BE A9 64              LDA  #$64
2256:  35C0 A0 A3              LDY  #$A3
2256:  35C2 20 1E AB           JSR  STROUT
2258:  35C5 A0 FF              LDY  #$FF
2259:  35C7 60                 RTS
2260:  35C8 9D 7A 03  L9       STA  SCHWARZE,X
2260:  35CB A9 0D              LDA  #$0D
2260:  35CD 20 D2 FF           JSR  CHROUT
2270:  35D0 20 CF FF           JSR  CHRIN
2280:  35D3 38                 SEC
2280:  35D4 E9 30              SBC  #$30
2280:  35D6 9D 8E 03           STA  WEISSE,X
2290:  35D9 A9 0D              LDA  #$0D
2300:  35DB 20 D2 FF           JSR  CHROUT
2300:  35DE 60                 RTS
2310:                 ;
2311:                 ; '1234'
2312:                 ;
2320:  35DF 31 32 33  TEXT     .ASC "1234"
2320:  35E3 00                 .BYT 0
3000:                 ;
3005:                 ; VERSCHIEBEN NACH $3400
3010:                 ;
3100:  35E4 A9 00              LDA  #0
3100:  35E6 85 6A              STA  ALT
3100:  35E8 85 6C              STA  NEU
3110:  35EA A9 04              LDA  #$04
3110:  35EC 85 6B              STA  ALT+1
3120:  35EE A9 34              LDA  #$34
3120:  35F0 85 6D              STA  NEU+1
3130:  35F2 A0 00     L10      LDY  #0
3140:  35F4 B1 6A     L11      LDA  (ALT),Y
3140:  35F6 91 6C              STA  (NEU),Y
3170:  35F8 C8                 INY
3180:  35F9 D0 F9              BNE  L11
3190:  35FB A5 6B              LDA  ALT+1
3200:  35FD C9 07              CMP  #$07
3210:  35FF F0 07              BEQ  AUFGEHTS
3220:  3601 E6 6B              INC  ALT+1
3220:  3603 E6 6D              INC  NEU+1
3230:  3605 18                 CLC
3230:  3606 90 EA              BCC  L10
3240:  3608 4C 00 34  AUFGEHTS JMP  $3400     ; PROGRAMM-START
```

Listing 2. Der Quell-Code des »Superhirn II« (Schluß)

Anmerkung: Ursprünglich war das Programm als »Bildschirmseite« konzipiert. Die Verschieberoutine ab Zeile 3000 wurde benötigt, um das Programm aus dem Bildschirmspeicher an seinen »Arbeitsplatz« zu bringen.

Ein schneller »Drawline«-Algorithmus

Im folgenden wird eine Möglichkeit vorgestellt, schnell und einfach eine Strecke, die durch ihre beiden Endpunkte gegeben ist, zu plotten. Als Ausgabegerät können Bildschirm, Drucker und Plotter eingesetzt werden.

Der Algorithmus wird in einer Basic- (Listing 1) und einer Assembler-Version (Listing 2) für einen 6510-Mikroprozessor mit den Adressen für den Commodore 64 beschrieben. Die Programme können, da sie im Aufbau einfach sind und erklärt werden, ohne große Probleme für andere Systeme beziehungsweise andere Sprachen umgewandelt werden. Das Maschinenprogramm (Listing 3) geben Sie bitte mit dem MSE ein.

Ein Rasterbildschirm setzt sich aus einzelnen Punkten, die gesetzt oder auch gelöscht sein können, zusammen. Der Abstand der Punkte voneinander ist in der Richtung der Achsen immer gleich und jeweils eine Schrittweite groß (Bild 1). Um eine Gerade zwischen den Punkten P0 und P1 zu ziehen, muß daher schrittweise berechnet werden, welcher Punkt der Ideallinie (Bild 2) am nächsten ist und daher gesetzt werden muß.

Geschwindigkeitsvorteile durch einfache Berechnungen

Für diese Berechnungen gibt es verschiedene Möglichkeiten, doch sind sie meistens mit Multiplikationen und Divisionen in der Approximationsschleife verbunden und daher weder schnell noch einfach zu programmieren. Der hier vorgestellte Algorithmus verwendet dagegen nur eine Division und in der Schleife nur mehr Addition, Subtraktion und eine Vergleichsoperation. Da die Schleifenoperationen außerdem nur mehr an Integerzahlen durchzuführen sind, ist er besonders schnell, und er läßt sich auch einfach programmieren. Zur Erklärung soll eine Gerade mit einer Steigung zwischen 0 und 1(0 bis 45 Grad) betrachtet werden.

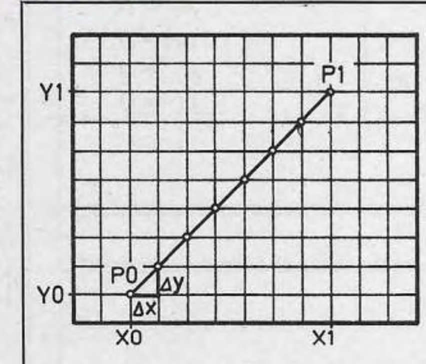
Wie man in Bild 2 unschwer erkennen kann, ist der Abstand der Punkte P0 und P1 entlang der X-Achse gleich der Anzahl der zu setzenden Punkte. DX = X1 - X0, das heißt, es sind DX-Approximationen durchzuführen, um die Gerade zu zeichnen. Für jeden folgenden Punkt ist als X0 und IX (= 1) zu erhöhen, während Y0 gleichbleibt (IY = 0) oder ebenfalls um 1 (IY =1) erhöht wird. Der Abstand der beiden Punkte entlang der Y-Achse ist demnach: DY = Y1 - Y0. Es bleibt also nur mehr festzustellen, wann IY = 0, beziehungsweise IY = 1 zu sein hat. Dazu wird vor Beginn der Schleife ein Approxima-

Name	Adressen						Beschreibung
	2001	3032	8032	VC20	C 64	610/710	
CHKCOM	CE11	CDF8	BEF5	CEFD	AEFD	9730	Prüfe ob nächstes Zeichen im Basic-Text ein Komma ist, wenn nicht gebe »SYNTAX ERROR« aus
GETCOR	D6C4	D6C6	C921	D7EB	B7EB	B4E5	Holt die Koordinaten eines Punktes aus dem Basic-Text. Die Routine wertet auch Ausdrücke aus. Die X-Koordinate wird als 2-Byte-Wert in X0, die Y-Koordinate als Byte im X-Register übergeben.
X0	0066	0014	0014	0014	0014	0011	Hier wird die X-Koordinate von GETCOR abgelegt.
PLOT	–	–	–	–	–	–	Diese Routine ist keine Betriebssystemroutine. Sie übernimmt die Koordinaten von X0 und dem X-Register.

Tabelle 1. Benützte Subroutinen und Übergabespeicherzellen

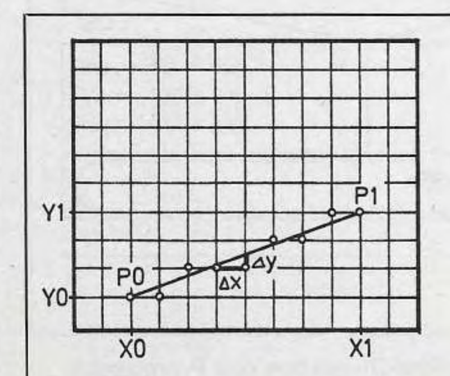
Name	Beschreibung
X0, Y0	Koordinaten des ersten Punktes
X1, Y1	Koordinaten des zweiten Punktes
CT	Schleifenzähler
IX	Inkrement oder Dekrement für X0 (-1,0,+1)
IY	Inkrement oder Dekrement für Y0 (-1,0,+1)
AX	wie IX für Steigungen > 1
AY	wie IY für Steigungen > 1
DX	Entfernung der Punkte entlang der X-Achse (= Anzahl der Punkte)
DY	Entfernung der Punkte entlang der Y-Achse
OF	Approximationsvariable zur Bestimmung ob Y0 gleichbleibt

Tabelle 2. Die verwendeten Variablen



DX = X1 - X0
DY = Y1 - Y2
DX = DY
Δx = 1
IX = Δx
Δy = 1
IY = Δy

Bild 1. Eine Strecke mit der Steigung 1 dargestellt auf dem Bildschirm



DX = X1 - X0
DY = Y1 - Y0
DX DY
Δx = 1
IX = Δx
0 ≤ Δy ≤ 1
IY = 0 oder 1

Bild 2. Eine Strecke mit der Steigung 1 und den optimierten Rasterpunkten

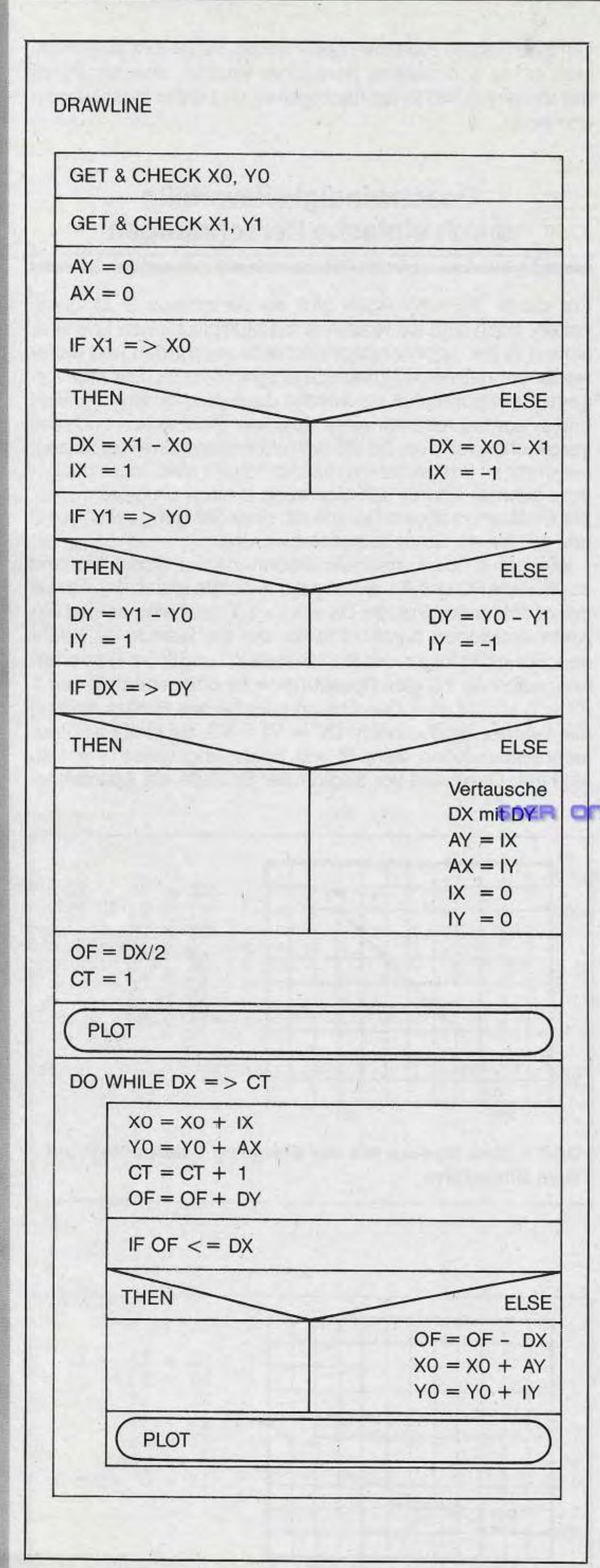


Bild 3. Nassi-Shneiderman-Diagramm des Programms »Draw«. Durch dieses Programm kann sehr schnell eine Strecke, die durch ihre beiden Endpunkte gegeben ist, berechnet werden.

tionswert OF berechnet. Da beim Idealfall für eine Steigung von 1 (Bild 1) IY = IX, das heißt, immer 1 ist, und DX = DY ist, wird OF = DX/2.

Zu OF wird für jeden neuen Punkt DY addiert. Solange OF kleiner als DX bleibt, bleibt IY = 0, wird OF gleich oder größer, so wird IY = 1, das heißt, Y0 um 1 erhöht. Damit diese Abfrage auch für die folgenden Punkte möglich ist, muß OF um DX vermindert werden.

Rasterpunkte optimieren

Wenn DX-Punkte gesetzt worden sind, sind alle Punkte der Geraden berechnet und die Schleife kann verlassen werden.

Soweit also für eine Gerade mit positiver Steigung zwischen 0 und 45 Grad. Wie sieht es aber aus, wenn zum Beispiel X1 kleiner X0 ist? Nun, dann wird eben IX = -1, X0 also für jeden Approximationsschritt um 1 erniedrigt. Und wie sieht es für eine Steigung größer 1 aus? Auch dieses Problem läßt sich einfach lösen. Es werden für die Rechnung einfach die beiden Achsen vertauscht.

Der Ablauf beider Programme ist im Nassi-Shneiderman-Diagramm (Bild 3) dargestellt. Zum leichteren Vergleich sind alle Variablenbezeichnungen in den Programmen identisch.

Die Basic-Version gibt nur die Koordinaten der berechneten Punkte aus, da hierfür keine »Setze-Punkt«-Routine geschrieben wurde. Man kann aber die Wirkungsweise des Algorithmus schön verfolgen.

Das Assemblerprogramm ist, wie schon oben gesagt, für den C 64 geschrieben mit einer möglichen Auflösung von 320 x 200 Punkten. Dadurch können die Werte für Y in einem Byte untergebracht werden, während für X zwei Byte benötigt werden. OF, DX und der Schleifenzähler CT benötigen deshalb ebenfalls 2 Byte. Die Länge der Werte muß beim Umstricken für ein anderes System berücksichtigt werden, da alle durchzuführenden Operationen dementsprechend 1 oder 2 Byte lang sind.

In dieser Assemblerversion werden nur 2 Subroutinen aus dem Betriebssystem verwendet. Sie sind in Tabelle 1 beschrieben. Zusätzlich sind die Adressen für die anderen Commodore Computer angegeben. Die Subroutine »PLOT« muß, wenn sie sich im Betriebssystem wie bei den Commodoresystemen nicht findet, extra geschrieben werden. Für den C 64 findet man im 64'er geeignete Programme. Die Routine CHKCOM überprüft, ob das nächste Zeichen des Basictextes (oder Eingabepuffers) ein Komma ist. Ist dies nicht der Fall, kehrt das Unterprogramm nicht zurück, sondern gibt »syntax error« aus und bricht das Programm ab. Beim Vorfinden eines Kommas wird das dem Komma folgende Zeichen in den Akku geholt. Ist dieses Zeichen eine Ziffer, dann wird übrigens das Carryflag auf »1« gesetzt. GETCOR holt zuerst eine Integerzahl (Übergabe in $14 und $15), prüft anschließend auf Komma, und holt dann das folgende Byte aus dem Text. Die Übergabe erfolgt im X-Register. Diese Routine ist wie geschaffen für die Eingabe von Koordinaten, da auch Formeln und Variablenwerte übernommen werden.

Kein Problem: Variablen

In Tabelle 2 sind alle verwendeten Variablen aufgelistet und beschrieben. Aufgerufen wird diese Version mit SYS aaaa,X0,Y0,X1,Y1, wobei aaaa die Startadresse der Routine, X0/Y0 und X1/Y1 die Koordinaten der beiden Punkte sind. Für die Koordinaten können auch Ausdrücke verwendet werden, da die Betriebssystemroutine »GETCOR« auch Ausdrücke auswertet.

(Michael Bauer/og)

```
100 REM               DRAWLINE               <006>
110 REM DIESE ROUTINE BERECHNET DIE KOORDI
    NATEN                                    <151>
120 REM DER PUNKTE AUF EINER STRECKE, DIE
    DURCH                                    <124>
130 REM DIE ENDPUNKTE GEGEBEN IST. MIT EIN
    ER GE-                                   <028>
140 REM EIGNETEN 'SETPOINT'-ROUTINE KANN D
    ER                                       <098>
150 REM BILDSCHIRM ODER EIN PLOTTER ANGEST
    EUERT                                    <184>
160 REM WERDEN. DIE GRENZEN DER WERTE FUER
     X UND                                   <093>
170 REM Y ENTSPRECHEN DEN WERTEN FUER DEN
    HI-RES                                   <165>
180 REM BILDSCHIRM DES COMMODORE 64.         <165>
190 REM                                      <252>
200 REM DAS ORIGINAL DIESES PROGRAMMES STA
    MMT VON                                  <131>
210 REM MIKE HIGGINS ERSCHIENEN IN BYTE 19
    81 IM                                    <129>
220 REM HEFT 8 AUF DEN SEITEN 414 - 416      <084>
230 REM                                      <036>
240 REM                                      <046>
250 INPUT"KOORDINATEN 1. PUNKT";X0,Y0        <024>
260 IF X0>319 OR Y0>199 THEN 250             <158>
270 IF X0<0 OR Y0<0 THEN 250                 <010>
280 INPUT"KOORDINATEN 2. PUNKT";X1,Y1        <203>
290 IF X1>319 OR Y1>199 THEN 280             <224>
300 IF X1<0 OR Y1<0 THEN 280                 <251>
310 REM SETPAR                               <244>
320 S1=0:S2=1:A1=1:A2=0                      <070>
330 IF X1=>X0 THEN D1=X1-X0:GOTO 360         <158>
340 A1=-1                                    <208>
350 D1=X0-X1                                 <203>
360 IF Y1=>Y0 THEN D2=Y1-Y0:GOTO 390         <206>
370 D2=Y0-Y1                                 <008>
380 S2=-1                                    <072>
390 IF D1=>D2 THEN 470                       <220>
400 N1=D1:REM VERTAUSCHE D1 UND D2           <225>
410 D1=D2                                    <117>
420 D2=N1                                    <136>
430 S1=A1                                    <037>
440 A1=0                                     <121>
450 A2=S2                                    <123>
460 S2=0                                     <221>
470 D3=D1/2                                  <137>
480 N1=1:REM SCHLEIFENZAEHLER                <014>
490 GOTO 580:REM PLOTTE URSPRUNGSPUNKT       <057>
500 X0=X0+A1                                 <018>
510 Y0=Y0+A2                                 <066>
520 D3=D3+D2                                 <050>
530 N1=N1+1                                  <158>
540 IF D3<=D1 THEN 580                       <255>
550 D3=D3-D1                                 <206>
560 X0=X0+S1                                 <098>
570 Y0=Y0+S2                                 <146>
580 PRINT X0,Y0                              <102>
590 IF D1=>N1 THEN 500                       <118>
600 END                                      <094>
```

Listing 1. Das Basic-Programm »Drawline«


```
PROGRAMM : DRAW                 C000 C34B
-----------------------------------------

C000 : 4C 2D C0 4C 44 C0 4C 5B  CF
C008 : C0 4C 78 C0 4C 92 C0 4C  19
C010 : 95 C0 4C 80 C1 4C 83 C1  38
C018 : 4C AB C1 4C AB C1 4C CE  4A
C020 : C1 4C D1 C1 4C 60 C1 4C  1B
C028 : 48 C1 4C 70 C1 AD 11 D0  E1
C030 : 8D 5B C3 AD 18 D0 8D 5C  08
C038 : C3 A9 3B 8D 11 D0 A9 18  BF
C040 : 8D 18 D0 60 A0 00 A9 20  0A
C048 : 84 FD 85 FE 98 91 FD C8  AC
C050 : D0 FB E6 FE A5 FE C9 40  B1
C058 : D0 F2 60 20 FD AE 20 9E  D0
C060 : B7 A0 00 A9 04 84 FD 85  04
C068 : FE 8A 91 FD C8 D0 FB E6  A0
C070 : FE A5 FE C9 08 D0 F2 60  CD
C078 : A0 00 A9 20 84 FD 85 FE  D3
C080 : B1 FD 49 FF 91 FD C8 D0  50
C088 : F7 E6 FE A5 FE C9 40 D0  47
C090 : EF 60 A9 00 2C A9 80 85  37
C098 : 97 20 FC C0 B0 F3 8A 4A  00
C0A0 : 4A 4A A8 B9 2F C1 8D 57  56
C0A8 : C3 B9 16 C1 8D 58 C3 8A  C5
C0B0 : 29 07 18 6D 57 C3 8D 57  89
C0B8 : C3 A5 14 29 F8 8D 59 C3  61
C0C0 : 18 A9 00 6D 57 C3 85 FD  00
C0C8 : A9 20 6D 58 C3 85 FE 18  7C
C0D0 : A5 FD 6D 59 C3 85 FD A5  A6
C0D8 : FE 65 15 85 FE A5 14 29  3F
C0E0 : 07 49 07 AA A9 01 CA 30  D1
C0E8 : 03 0A D0 FA A0 00 24 97  4D
C0F0 : 10 05 49 FF 31 FD 2C 11  AB
C0F8 : FD 91 FD 60 20 FD AE 20  36
C100 : EB B7 E0 C8 B0 0C A5 15  44
C108 : C9 01 90 07 D0 04 A4 14  3F
C110 : C0 40 60 A4 14 60 00 01  E3
C118 : 02 03 05 06 07 08 0A 0B  8D
C120 : 0C 0D 0F 10 11 12 14 15  95
C128 : 16 17 19 1A 1B 1C 1E 00  5F
C130 : 40 80 C0 00 40 80 C0 00  EC
C138 : 40 80 C0 00 40 80 C0 00  F4
C140 : 40 80 C0 00 40 80 C0 00  FC
C148 : 20 FD AE 20 D4 E1 A2 00  FD
C150 : A0 40 A9 00 85 FD A9 20  AA
C158 : 85 FE A9 FD 20 D8 FF 60  10
C160 : 20 FD AE 20 D4 E1 A9 61  F4
C168 : 85 B9 A9 00 20 D5 FF 60  A6
C170 : AD 5B C3 8D 11 D0 AD 5C  74
C178 : C3 8D 18 D0 20 44 E5 60  9E
C180 : A9 00 2C A9 80 85 97 20  3C
C188 : FD AE 20 9E B7 E0 C8 B0  BF
C190 : EE 8E 4B C3 86 FD A0 3F  6A
C198 : 8C 4C C3 A0 01 8C 4D C3  80
C1A0 : 88 84 15 84 14 4C FA C1  53
C1A8 : A9 00 2C A9 80 85 97 20  64
C1B0 : FD AE 20 8A AD 20 F7 B7  89
C1B8 : 20 08 C1 B0 C2 8D 4D C3  B8
C1C0 : 8C 4C C3 A2 C7 86 FD A0  A1
C1C8 : 00 8C 4B C3 F0 D7 A9 00  CE
C1D0 : 2C A9 80 85 97 20 FC C0  92
C1D8 : B0 E1 8E 4B C3 8D 52 C3  FF
C1E0 : 8C 51 C3 20 FC C0 B0 D3  4A
C1E8 : 86 FD 8C 4C C3 AC 51 C3  88
C1F0 : 84 14 8D 4D C3 AD 52 C3  06
C1F8 : 85 15 A0 01 8C 56 C3 8C  F4
C200 : 53 C3 8C 51 C3 88 8C 52  D9
C208 : C3 8C 55 C3 8C 54 C3 88  6B
C210 : AD 4D C3 C5 15 90 09 D0  A9
C218 : 1E AD 4C C3 C5 14 B0 17  86
C220 : 38 A5 14 ED 4C C3 8D 4F  A5
C228 : C3 A5 15 ED 4D C3 8D 50  8B
C230 : C3 8C 56 C3 4C 48 C2 38  CA
C238 : AD 4C C3 E5 14 8D 4F C3  2B
C240 : AD 4D C3 E5 15 8D 50 C3  C8
C248 : A5 FD CD 4B C3 B0 0F 38  37
C250 : AD 4B C3 E5 FD 8D 4E C3  5D
C258 : 8C 53 C3 4C 64 C2 ED 4B  B3
C260 : C3 8D 4E C3 AD 50 C3 D0  04
C268 : 24 AD 4F C3 CD 4E C3 B0  6F
C270 : 1C AE 4E C3 8D 4E C3 8E  67
C278 : 4F C3 AD 56 C3 8D 54 C3  60
C280 : AD 53 C3 8D 55 C3 C8 8C  29
C288 : 56 C3 8C 53 C3 AD 50 C3  C0
C290 : 4A 8D 4D C3 AD 4F C3 6A  A6
C298 : 8D 4C C3 4C 31 C3 AD 56  5A
C2A0 : C3 30 0E 18 65 14 85 14  37
C2A8 : A5 15 69 00 85 15 4C BE  E2
C2B0 : C2 38 A5 14 E9 01 85 14  5F
C2B8 : A5 15 E9 00 85 15 18 AD  1F
C2C0 : 4B C3 6D 55 C3 8D 4B C3  50
C2C8 : 18 AD 4C C3 6D 4E C3 8D  B6
C2D0 : 4C C3 AD 4D C3 69 00 8D  B5
C2D8 : 4D C3 EE 51 C3 D0 03 EE  99
C2E0 : 52 C3 AD 4D C3 CD 50 C3  9C
C2E8 : 90 47 D0 08 AD 4F C3 CD  51
C2F0 : 4C C3 B0 3D 38 AD 4C C3  9B
C2F8 : ED 4F C3 8D 4C C3 AD 4D  64
C300 : C3 ED 50 C3 8D 4D C3 AD  F4
C308 : 54 C3 30 0E 18 65 14 85  14
C310 : 14 A5 15 69 00 85 15 4C  82
C318 : 27 C3 38 A5 14 E9 01 85  83
C320 : 14 A5 15 E9 00 85 15 18  3A
C328 : AD 4B C3 6D 53 C3 8D 4B  39
C330 : C3 AE 4B C3 20 9E C0 AD  EB
C338 : 52 C3 CD 50 C3 90 09 AD  29
C340 : 4F C3 CD 51 C3 B0 01 60  95
C348 : 4C 9E C2 FF 00 FF 00 FF  94
```

Listing 3. Das Maschinen-Programm »Draw« geben Sie bitte mit dem MSE ein.

```
212 zeilen 37 symbole
0000          1 ;algorithmus basiert auf einen artikel von mike
0000          2 ;higgins erschienen in der byte august 1981 s.414-416
0000          3 ;
0000          4 ;das programm fuer den cbm 64 wurde von
0000          5 ;michael bauer    aindorferstr. 86   8000 muenchen 21
0000          6 ;geschrieben.
0000          7 ;
0014          8 x0     =   $14        ;x-koordinate
00fd          9 y1     =   $fd        ;y-koordinate
aefd         10 chkcom =   $aefd      ;prueft ob ein komma folgt,
0000         11 ;                      wenn ja, holt das naechste
0000         12 ;                      zeichen, wenn nein -> syntax
0000         13 ;                      error
b7eb         14 getcor =   $b7eb      ;holt eine adresse (= 2 byte)
0000         15 ;                      und ein byte. die adresse wird
0000         16 ;                      in x0 und x0+1, das byte im
0000         17 ;                      x-register uebergeben.
0000         18 plot   =   $0000      ;die adresse dieser routine
0000         19 ;                      muss hier eingesetzt werden.
0000         20 ;                      plot muss in der lage sein
0000         21 ;                      einen punkt, dessen x und y
0000         22 ;                      koordinaten uebergeben werden,
0000         23 ;                      zu setzen.
0000         24 ;
c178         25        * = $c178
c178         26 ;
c178 60      27 ill    rts
c179         28 ;********* einsprung: aufruf  sys49529,x0,y0,x1,y1
c179 20efc2  29 drwlin jsr gcoord     ;hole koordinaten 1. punkt
c17c b0fa    30        bcs ill        ;ignoriere wert wenn ausserhalb
c17e 8e09c3  31        stx y0
c181 8d10c3  32        sta ct+1
c184 8c0fc3  33        sty ct
c187 20efc2  34        jsr gcoord     ;hole koordinaten 2. punkt
c18a b0ec    35        bcs ill        ;ignoriere wert wenn ausserhalb
c18c 86fd    36        stx y1
c18e 8c0ac3  37        sty x1
c191 ac0fc3  38        ldy ct
c194 8414    39        sty x0
c196 8d0bc3  40        sta x1+1
c199 ad10c3  41        lda ct+1
c19c 8515    42        sta x0+1
c19e a001    43 setpar ldy @$01       ;(yr) = 1
c1a0 8c14c3  44        sty ix
c1a3 8c11c3  45        sty iy
c1a6 8c0fc3  46        sty ct
c1a9 88      47        dey            ;(yr) = 0
c1aa 8c10c3  48        sty ct+1
c1ad 8c13c3  49        sty ax
c1b0 8c12c3  50        sty ay
c1b3 88      51        dey            ;(yr) = - 1
c1b4         52 ;                      berechne dx
c1b4 ad0bc3  53 begin  lda x1+1       ;x1 => x0 ?
c1b7 c515    54        cmp x0+1
c1b9 9009    55        bcc draw01     ;nein ->
c1bb d01e    56        bne draw02     ;ja ->
c1bd ad0ac3  57        lda x1
c1c0 c514    58        cmp x0
c1c2 b017    59        bcs draw02     ;ja ->
c1c4 38      60 draw01 sec            ;dx = x0 - x1
c1c5 a514    61        lda x0
c1c7 ed0ac3  62        sbc x1
c1ca 8d0dc3  63        sta dx
c1cd a515    64        lda x0+1
c1cf ed0bc3  65        sbc x1+1
c1d2 8d0ec3  66        sta dx+1
c1d5 8c14c3  67        sty ix         ;ix = -1
c1d8 4cecc1  68        jmp draw03
c1db 38      69 draw02 sec            ;dx = x1 - x0
c1dc ad0ac3  70        lda x1
c1df e514    71        sbc x0
c1e1 8d0dc3  72        sta dx
c1e4 ad0bc3  73        lda x1+1
c1e7 e515    74        sbc x0+1
c1e9 8d0ec3  75        sta dx+1
c1ec         76 ;                      berechne dy
c1ec a5fd    77 draw03 lda y1         ;y1 => y0 ?
c1ee cd09c3  78        cmp y0
c1f1 b00f    79        bcs draw04     ;ja ->
c1f3 38      80        sec            ;dy = y0 - y1
c1f4 ad09c3  81        lda y0
c1f7 e5fd    82        sbc y1
c1f9 8d0cc3  83        sta dy
c1fc 8c11c3  84        sty iy         ;iy = -1
c1ff 4c08c2  85        jmp draw05
c202 ed09c3  86 draw04 sbc y0         ;dy = y1 - y0
c205 8d0cc3  87        sta dy
c208 ad0ec3  88 draw05 lda dx+1       ;dx < dy ?
c20b d024    89        bne draw07     ;nein ->
c20d ad0dc3  90        lda dx
c210 cd0cc3  91        cmp dy
c213 b01c    92        bcs draw07     ;nein ->
c215 ae0cc3  93        ldx dy         ;vertausche die achsen
c218 8d0cc3  94        sta dy
c21b 8e0dc3  95        stx dx
c21e ad14c3  96        lda ix         ;ay = ix
c221 8d12c3  97        sta ay
c224 ad11c3  98        lda iy         ;ax = iy
c227 8d13c3  99        sta ax
c22a c8     100        iny            ;(yr) = 0
c22b 8c14c3 101        sty ix         ;ix = 0
c22e 8c11c3 102        sty iy         ;iy = 0
c231 ad0ec3 103 draw07 lda dx+1       ;of = dx / 2
c234 4a     104        lsr a
c235 8d0bc3 105        sta of+1       ;= 0
c238 ad0dc3 106        lda dx
c23b 6a     107        ror a
c23c 8d0ac3 108        sta of
c23f 4cd5c2 109        jmp plotit     ;plotte 1. punkt
c242        110 ;********* approximationsschleife
c242 ad14c3 111 drwlop lda ix         ;ix = -1 ?
c245 300e   112        bmi draw08     ;ja -> weiter
c247 18     113        clc            ;x0 = x0 + ix
c248 6514   114        adc x0
c24a 8514   115        sta x0
c24c a515   116        lda x0+1
c24e 6900   117        adc @$00
c250 8515   118        sta x0+1
c252 4c62c2 119        jmp draw11
c255 38     120 draw08 sec            ;x0 = x0 - ix
c256 a514   121        lda x0
c258 e901   122        sbc @$01
c25a 8514   123        sta x0
c25c a515   124        lda x0+1
c25e e900   125        sbc @$00
c260 8515   126        sta x0+1
c262 18     127 draw11 clc            ;y0 = y0 + ax
c263 ad09c3 128        lda y0
c266 6d13c3 129        adc ax
c269 8d09c3 130        sta y0
c26c 18     131        clc            ;of = of + dy
c26d ad0ac3 132        lda of
c270 6d0cc3 133        adc dy
c273 8d0ac3 134        sta of
c276 ad0bc3 135        lda of+1
c279 6900   136        adc @$00
c27b 8d0bc3 137        sta of+1
c27e ee0fc3 138        inc ct         ;ct = ct + 1
c281 d003   139        bne draw06
c283 ee10c3 140        inc ct+1
c286 ad0bc3 141 draw06 lda of+1       ;of <= dx ?
c289 cd0ec3 142        cmp dx+1
c28c 9047   143        bcc plotit     ;ja -> zeichne punkt
c28e d008   144        bne draw09
c290 ad0dc3 145        lda dx
c293 cd0ac3 146        cmp of
c296 b03d   147        bcs plotit     ;ja -> zeichne punkt
c298 38     148 draw09 sec            ;of = of - dx
c299 ad0ac3 149        lda of
c29c ed0dc3 150        sbc dx
c29f 8d0ac3 151        sta of
c2a2 ad0bc3 152        lda of+1
c2a5 ed0ec3 153        sbc dx+1
c2a8 8d0bc3 154        sta of+1
c2ab ad12c3 155        lda ay         ;ay = -1 ?
c2ae 300e   156        bmi draw10     ;ja -> weiter
c2b0 18     157        clc            ;x0 = x0 + ay
c2b1 6514   158        adc x0
c2b3 8514   159        sta x0
c2b5 a515   160        lda x0+1
c2b7 6900   161        adc @$00
c2b9 8515   162        sta x0+1
c2b[illegible]cbc2 163        jmp draw12
c2be 38     164 draw10 sec            ;x0 = x0 - ay
c2bf a514   165        lda x0
c2c1 e901   166        sbc @$01
c2c3 8514   167        sta x0
c2c5 a515   168        lda x0+1
c2c7 e900   169        sbc @$00
c2c9 8515   170        sta x0+1
c2cb 18     171 draw12 clc            ;y0 = y0 + iy
c2cc ad09c3 172        lda y0
c2cf 6d11c3 173        adc iy
c2d2 8d09c3 174        sta y0
c2d5 ae09c3 175 plotit ldx y0         ;uebergebe y0 im (xr) an plot
c2d8 200000 176        jsr plot       ;zeichne punkt
c2db ad10c3 177        lda ct+1       ;ct <= dx ?
c2de cd0ec3 178        cmp dx+1
c2e1 9009   179        bcc nexpnt     ;nein -> naechster punkt
c2e3 ad0dc3 180        lda dx
c2e6 cd0fc3 181        cmp ct
c2e9 b001   182        bcs nexpnt
c2eb 60     183        rts            ;linie ist fertig gezeichnet
c2ec 4c42c2 184 nexpnt jmp drwlop
c2ef        185 ;
c2ef 20fdae 186 gcoord jsr chkcom     ;hole und pruefe parameter
c2f2 20ebb7 187        jsr getcor
c2f5 e0c8   188        cpx @200       ;y > 200 ?
c2f7 b00c   189        bcs finish
c2f9 a515   190        lda x0+1       ;x > 320 ?
c2fb c901   191        cmp @$01
c2fd 9007   192        bcc finis      ;< 1
c2ff d004   193        bne finish     ;> 1
c301 a414   194        ldy x0         ;= 1
c303 c040   195        cpy @$40
c305 60     196 finish rts
c306 a414   197 finis  ldy x0
c308 60     198        rts
c309        199 ;
c309        200 ;arbeitsvariable
c309        201 ;
c309        202 y0     =   *          ;y-koordinate
c30a        203 x1     =   y0+1       ;x-koordinate
c30a        204 of     =   x1         ;approximationswert
c30c        205 dy     =   of+2       ;abstand y1 zu y0
c30d        206 dx     =   dy+1       ;anzahl der punkte
c30f        207 ct     =   dx+2       ;schleifenzaehler
c311        208 iy     =   ct+2       ;inkrement
c312        209 ay     =   iy+1       ;wie iy
c313        210 ax     =   ay+1       ;wie ix
c314        211 ix     =   ax+1       ;inkrement
c309        212 .end
x0     = 0014    y1     = 00fd    chkcom = aefd    getcor = b7eb
plot   = 0000    ill    = c178    drwlin = c179    setpar = c19e
begin  = c1b4    draw01 = c1c4    draw02 = c1db    draw03 = c1ec
draw04 = c202    draw05 = c208    draw07 = c231    drwlop = c242
draw08 = c255    draw11 = c262    draw06 = c286    draw09 = c298
draw10 = c2be    draw12 = c2cb    plotit = c2d5    nexpnt = c2ec
gcoord = c2ef    finish = c305    finis  = c306    y0     = c309
x1     = c30a    of     = c30a    dy     = c30c    dx     = c30d
ct     = c30f    iy     = c311    ay     = c312    ax     = c313
ix     = c314
```

Listing 2. Quellcode zu »Draw«.

Befehlssatz des 6510

Hier finden Sie, alphabetisch geordnet, eine Auflistung aller bekannten Befehle des C64-Prozessors. Dazu gehören auch die »illegalen Opcodes«.

Zuerst ein Wort zu den illegalen Opcodes, die in Tabelle 1 enthalten sind:

Seit Erscheinen des C 64 vor ungefähr drei Jahren sind einige verschiedene Versionen des Prozessors 6510 gebaut worden. Diese sind untereinander voll kompatibel, was den normalen Befehlssatz aus Tabelle 2 anbetrifft. Die illegalen Opcodes jedoch laufen nicht auf allen Versionen der CPU 6510. Welche Befehle auf welchem Computer eine korrekte Ausführung bewirken, läßt sich nur durch Ausprobieren feststellen. Äußerst hilfreich dabei ist der SMON aus dieser Ausgabe: Er zeigt einen illegalen Opcode nicht wie die meisten Maschinensprachmonitore durch drei Fragezeichen an, sondern disassembliert den Befehl mit den in Tabelle 1 genannten Abkürzungen. Ein vorangestelltes Sternchen (*) kennzeichnet bei SMON den Befehl als illegalen Opcode (zum Beispiel *AXS).

In Tabelle 3 finden Sie eine Übersicht über die in den beiden anderen Tabellen verwendeten Abkürzungen.

(tr)


```
■A11 : AND register with $11
Das X- beziehungsweise Y-Register wird mit $11 AND-
verknüpft und das Ergebnis X- beziehungsweise Y-
indiziert abgelegt

Addressierungsarten:
Assembler:   Hex-Code:   Abkürzung:   Byte:

A11 OP,X        9C          ABX        3
A11 OP,Y        9E          ABY        3

■AAX : AND akku with X-Register and store akku
Entspricht befehlsfolge:
AND zwischen Akku und X-Register
STA

Addressierungsarten:
Assembler:   Hex-Code:   Abkürzung:   Byte:

AAX #OP         8B          IM         2
AAX OP          87          ZP         2
AAX OP,Y        97          ZPY        2
AAX OP          8F          ABS        3
AAX (OP,X)      83          (OP,X)     2

■ASR : AND with akku and shift right
Entspricht Befehlsfolge:
AND
LSR

Addressierungsarten:
Assembler:   Hex-Code:   Abkürzung:   Byte:

ASR #OP         6B          IM         2

■ARR : AND with akku and rotate right
Entspricht Befehlsfolge:
AND
ROR

Addressierungsarten:
Assembler:   Hex-Code:   Abkürzung:   Byte:

ARR #OP         4b          IM         2

■AXS : AND akku and X-register and subtract from data
Der Wert wird von dem Ergebnis der AND-Verknüpfung
zwischen Akku und X-Register subtrahiert und in das X-
Register geschrieben.

Addressierungsarten:
Assembler:   Hex-Code:   Abkürzung:   Byte:

AXS #OP         CB          IM         2

■DCP : decrement and compare with akku
Entspricht Befehlsfolge:
DEC
CMP

Addressierungsarten:
Assembler:   Hex-Code:   Abkürzung:   Byte:

DCP OP          C7          ZP         2
DCP OP,X        D7          ZPX        2
DCP OP          CF          ABS        3
DCP OP,X        DF          ABX        3
DCP OP,Y        DB          ABY        3
DCP (OP,X)      C3          (ZP,X)     2
DCP (OP),Y      D3          (ZP),Y     2

■DOP : double NOP
Folgende Codes wirken wie der NOP-Befehl, sind aber
zwei Byte lang. Das zweite Byte wird dabei
übersprungen.

04, 14, 34, 44, 54, 64, 74, D4, F4, 80, 89, 93

■ISC : increment and subtract with carry
Entspricht Befehlsfolge:
INC
SBC

Addressierungsarten:
Assembler:   Hex-Code:   Abkürzung:   Byte:

ISC OP          E7          ZP         2
ISC OP,X        F7          ZPX        2
ISC OP          EF          ABS        3
ISC OP,X        FF          ABX        3
ISC OP,Y        FB          ABY        3
ISC (OP,X)      E3          (OP,X)     2
ISC (OP),Y      F3          (OP),Y     2

■KIL : killer codes
Folgende Codes bewirken einen Absturz des Prozessors,
dem auch mit einem RUN/STOP-RESTORE nicht mehr
beizukommen ist.

02, 12, 22, 32, 42, 52, 62, 72, 92, B2, D2, F2

■LAR : load akku, AND with stackregister, transfer
result to akku, X-register and stackregister
Entspricht Befehlsfolge:
LDA
AND
TAX
TXS

Addressierungsarten:
Assembler:   Hex-Code:   Abkürzung:   Byte:

LAR OB,Y        BB          ABY        3

■LAX : load to akku and X-register
Entspricht Befehlsfolge:
LDA
TAX

Addressierungsarten:
Assembler:   Hex-Code:   Abkürzung:   Byte:

LAX OP          A7          ZP         2
LAX OP,Y        B7          ZPY        2
LAX OP          AF          ABS        3
LAX OP,Y        BF          ABY        3
LAX (OP,X)      A3          (OP,X)     2
LAX (OP),Y      B3          (OP),Y     2

■NOP : no operation
Folgende Codes haben wie der Code $EA die NOP-Funktion:

1A, 3A, 5A, 7A, DA, FA
```

Tabelle 1. Die »illegalen Opcodes« des 6510-Prozessors

■ RLA : rotate left, AND with akku and store akku
Entspricht Befehlsfolge:
ROL
AND
STA

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:
RLA OP	27	ZP	2
RLA OP,X	37	ZPX	2
RLA OP	2F	ABS	3
RLA OP,X	3F	ABX	3
RLA OP,Y	3B	ABY	3
RLA (OP,X)	23	(OP,X)	2
RLA (OP),Y	33	(OP),Y	2

■ RRA : rotate right and add with carry
Entspricht Befehlsfolge:
ROR
ADC

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:
RRA OP	67	ZP	2
RRA OP,X	77	ZPX	2
RRA OP	6F	ABS	3
RRA OP,X	7F	ABX	3
RRA OP,Y	7B	ABY	3
RRA (OP,X)	63	(OP,X)	2
RRA (OP),Y	73	(OP),Y	2

■ SLO : shift left and OR with akku
Entspricht Befehlsfolge:
ASL
ORA

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:
SLO OP	07	ZP	2
SLO OP,X	17	ZPX	2
SLO OP	0F	ABS	3
SLO OP,X	1F	ABX	3
SLO OP,Y	1B	ABY	3
SLO (OP,X)	13	(OP,X)	2
SLO (OP),Y	03	(OP),Y	2

■ SRE : shift right and EOR with akku
Entspricht Befehlsfolge:
LSR
EOR

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:
SRE OP	47	ZP	2
SRE OP,X	57	ZPX	2
SRE OP	4F	ABS	3
SRE OP,X	5F	ABX	3
SRE OP,Y	5B	ABY	3
SRE (OP,X)	43	(OP,X)	2
SRE (OP),Y	53	(OP),Y	2

■ TOP : triple NOP
Folgende Codes wirken wie der NOP-Befehl, sind aber drei Byte lang. Das zweite und das dritte Byte wird dabei übersprungen.

0C, 1C, 3C, 5C, 7C, DC, FC

Tabelle 1. Die »illegalen Opcodes« des 6510-Prozessors (Schluß)

■ ADC : add with carry
addiere Adresseninhalt plus Carry-Flag zum Akkumulator

Flags: N Z C I D V
+ + + +

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
ADC #OP	69	IM	2	2
ADC OP	65	ZP	2	3
ADC OP,X	75	ZPX	2	4
ADC OP	6D	ABS	3	4
ADC OP,X	7D	ABX	3	4
ADC OP,Y	79	ABY	3	4
ADC (OP,X)	61	(ZP,X)	2	6
ADC (OP),Y	71	(ZP),Y	2	5

■ AND : AND akku
verknüpfe Speicher mit Akku durch logische UND

Flags: N Z C I D V
+ +

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
AND #OP	29	IM	2	2
AND OP	25	ZP	2	3
AND OP,X	35	ZPX	2	4
AND OP	2D	ABS	3	4
AND OP,X	3D	ABX	3	4
AND OP,Y	39	ABY	3	4
AND (OP,X)	21	(ZP,X)	2	6
AND (OP),Y	31	(ZP),Y	2	5

■ ASL : arithmetic shift left
schiebe Bits eines Speichers um eine Stelle nach links

Flags: N Z C I D V
+ + +

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
ASL	0A	Akku	1	2
ASL OP	06	ZP	2	5
ASL OP,X	16	ZPX	2	6
ASL OP	0E	ABS	3	6
ASL OP,X	1E	ABX	3	7

■ BCC : branch if carry clear
verzweige, falls das Übertragsbit gelöscht ist

Flags: N Z C I D V
keine

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
BCC OP	90	REL	2	2

■ BCS : branch if carry set
verzweige, falls das Übertragsbit gesetzt ist

Flags: N Z C I D V
keine

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
BCS OP	B0	REL	2	2

■ BEQ : branch if equal (to zero)
verzweige, falls das Ergebnis der letzten Operation gleich (Null) war

Flags: N Z C I D V
keine

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
BEQ OP	F0	REL	2	2

■ BIT : test bits
verknüpfe Speicher und Akku durch AND, setze entsprechende Flags (Akku wird nicht verändert !)

Flags: N Z C I D V
+ + +

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
BIT OP	24	ZP	2	3
BIT OP	2C	ABS	3	4

■ BMI : branch if minus
verzweige, falls das Ergebnis der letzten Operation kleiner Null war

Flags: N Z C I D V
keine

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
BMI OP	30	REL	2	2

Tabelle 2. Die Befehle des 6510-Prozessors



■ **BNE : branch if not equal (to zero)**
verzweige, falls das Ergebnis der letzten Operation ungleich (Null) war

Flags: N Z C I D V
keine

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
BNE OP	D0	REL	2	2

■ **BPL : branch if plus**
verzweige, falls das Ergebnis der letzten Operation größer Null war

Flags: N Z C I D V
keine

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
BPL OP	10	REL	2	2

■ **BRK : break**
Programmstop und Sprung über Breakpointer

Flags: N Z C I D V
+ (D)

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
BRK	00	–	1	7

■ **BVC : branch if overflow clear**
verzweige, falls das Überlaufsbit gelöscht ist

Flags: N Z C I D V
keine

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
BVC OP	50	REL	2	2

■ **BVS : branch if overflow set**
verzweige, falls das Überlaufsbit gesetzt ist

Flags: N Z C I D V
keine

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
BVS OP	70	REL	2	2

■ **CLC : clear carry**
lösche das Übertragsbit

Flags: N Z C I D V
+ (C)

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
CLC	18	–	1	2

■ **CLD : clear decimal mode**
lösche das Bit für den Dezimalmodus

Flags: N Z C I D V
+ (D)

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
CLD	D8	–	1	2

■ **CLI : clear interrupt flag**
lösche das Interruptbit (Interrupts nun erlaubt)

Flags: N Z C I D V
+ (I)

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
CLI	58	–	1	2

■ **CLV : clear overflow flag**
lösche das Überlaufbit

Flags: N Z C I D V
+ (V)

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
CLV	B8	–	1	2

■ **CMP : compare with akku**
vergleiche Speicher mit Akkuinhalt

Flags: N Z C I D V
+ + + (N Z C)

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
CMP #OP	C9	IM	2	2
CMP OP	C5	ZP	2	3
CMP OP,X	D5	ZPX	2	4
CMP OP	CD	ABS	3	4
CMP OP,X	DD	ABX	3	4
CMP OP,Y	D9	ABY	3	4
CMP (OP,X)	C1	(ZP,X)	2	6
CMP (OP),Y	D1	(ZP),Y	2	5

■ **CPX : compare with X-register**
vergleiche Speicherinhalt mit X-Register

Flags: N Z C I D V
+ + + (N Z C)

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
CPX #OP	C9	IM	2	2
CPX OP	E4	ZP	2	3
CPX OP	EC	ABS	3	4

■ **CPY : compare with Y-register**
vergleiche Speicherinhalt mit Y-Register

Flags: N Z C I D V
+ + + (N Z C)

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
CPY #OP	C0	IM	2	2
CPY OP	C4	ZP	2	3
CPY OP	CC	ABS	3	4

■ **DEC : decrement**
subtrahiere Eins von Speicherinhalt

Flags: N Z C I D V
+ + (N Z)

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
DEC OP	C6	ZP	2	5
DEC OP,X	D6	ZPX	2	6
DEC OP	CE	ABS	3	6
DEC OP,X	DE	ABX	3	7

■ **DEX : decrement X-register**
subtrahiere Eins vom Inhalt des X-Registers

Flags: N Z C I D V
+ + (N Z)

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
DEX	CA	–	1	2

■ **DEY : decrement Y-register**
subtrahiere Eins vom Inhalt des Y-Registers

Flags: N Z C I D V
+ + (N Z)

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
DEY	88	–	1	2

Tabelle 2. Die Befehle des 6510-Prozessors (Fortsetzung)

■ **EOR : exclusive-or**
verknüpfe Akku und Speicher durch logisches EXKLUSIV-ODER

Flags: N Z C I D V
+ +

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
EOR #OP	49	IM	2	2
EOR OP	45	ZP	2	3
EOR OP,X	55	ZPX	2	4
EOR OP	4D	ABS	3	4
EOR OP,X	5D	ABX	3	4
EOR OP,Y	59	ABY	3	4
EOR (OP,X)	41	(ZP,X)	2	6
EOR (OP),Y	51	(ZP),Y	2	5

■ **INC : increment**
addiere Eins zu Speicherinhalt

Flags: N Z C I D V
+ +

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
INC OP	E6	ZP	2	5
INC OP,X	F6	ZPX	2	6
INC OP	EE	ABS	3	6
INC OP,X	FE	ABX	3	7

■ **INX : increment X-register**
addiere Eins zu X-Registerinhalt

Flags: N Z C I D V
+ +

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
INX	E8	–	1	2

■ **INY : increment Y-register**
addiere Eins zu Y-Registerinhalt

Flags: N Z C I D V
+ +

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
INY	C8	–	1	2

■ **JMP : jump**
springe zu Adresse

Flags: N Z C I D V
keine

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
JMP OP	4C	ABS	3	3
JMP (OP)	6C	IND	3	5

■ **JSR : jump subroutine**
Springe in Unterprogramm

Flags: N Z C I D V
keine

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
JSR OP	20	ABS	3	6

■ **LDA : load akku**
schreibe Wert in Akku

Flags: N Z C I D V
+ +

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
LDA #OP	A9	IM	2	2
LDA OP	A5	ZP	2	3
LDA OP,X	B5	ZPX	2	4
LDA OP	AD	ABS	3	4
LDA OP,X	BD	ABX	3	4
LDA OP,Y	B9	ABY	3	4
LDA (OP,X)	A1	(ZP,X)	2	6
LDA (OP),Y	B1	(ZP),Y	2	5

■ **LDX : load X-register**
schreibe Wert ins X-Register

Flags: N Z C I D V
+ +

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
LDX #OP	A2	IM	2	2
LDX OP	A5	ZP	2	3
LDX OP,Y	B6	ZPY	2	4
LDX OP	AE	ABS	3	4
LDX OP,Y	BE	ABY	3	4

■ **LDY : load Y-register**
schreibe Wert ins Y-Register

Flags: N Z C I D V
+ +

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
LDY #OP	A0	IM	2	2
LDY OP	A4	ZP	2	3
LDY OP,X	B4	ZPX	2	4
LDY OP	AC	ABS	3	4
LDY OP,X	BC	ABX	3	4

■ **LSR : logical shift right**
bitweises Rechtsschieben eines Speicherinhalts (Bit 0 wird ins Carry-Flag geschoben, Bit 7 wird auf Null gesetzt)

Flags: N Z C I D V
+ + +

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
LSR	4A	Akku	1	2
LSR OP	46	ZP	2	5
LSR OP,X	56	ZPX	2	6
LSR OP	4E	ABS	3	6
LSR OP,X	5E	ABX	3	7

■ **NOP : no operation**
keine Ausführung (Dummy-Befehl)

Flags: N Z C I D V
keine

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
NOP	EA	–	1	2

■ **ORA : OR akku**
verknüpfe Speicherinhalt und Akku durch logisches ODER

Flags: N Z C I D V
+ +

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
ORA #OP	09	IM	2	2
ORA OP	05	ZP	2	3
ORA OP,X	15	ZPX	2	4
ORA OP	0D	ABS	3	4
ORA OP,X	1D	ABX	3	4
ORA OP,Y	19	ABY	3	4
ORA (OP,X)	01	(ZP,X)	2	6
ORA (OP),Y	11	(ZP),Y	2	5

■ **PHA : push akku**
schiebe Akkuinhalt auf Stack

Flags: N Z C I D V
keine

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
PHA	48	–	1	3

■ **PHP : push processor-status**
schiebe Statusregister auf Stack

Tabelle 2. Die Befehle des 6510-Prozessors (Fortsetzung)

Flags: N Z C I D V
keine

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
PHP	08	–	1	3

■ **PLA : pull akku**
lade Akku mit oberstem Stackbyte

Flags:

N	Z	C	I	D	V
+	+				

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
PLA	68	–	1	3

■ **PLP : pull processor-status**
lade Statusregister mit oberstem Stackbyte

Flags:

N	Z	C	I	D	V
+	+	+	+	+	+

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
PLP	28	–	1	4

■ **ROL : rotate left**
rotiere Speicherinhalt um ein Bit nach links (Bit 7 kommt ins Carryflag, Inhalt des Carry-Flags kommt ins Bit Ø)

Flags:

N	Z	C	I	D	V
+	+	+			

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
ROL	2A	Akku	1	2
ROL OP	26	ZP	2	5
ROL OP,X	36	ZPX	2	6
ROL OP	2E	ABS	3	6
ROL OP,X	3E	ABX	3	7

■ **ROR : rotate right**
rotiere Speicherinhalt um ein Bit nach rechts (Bit Ø kommt ins Carryflag, Inhalt des Carryflags kommt ins Bit 7)

Flags:

N	Z	C	I	D	V
+	+	+			

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
ROR	6A	Akku	1	2
ROR OP	66	ZP	2	5
ROR OP,X	76	ZPX	2	6
ROR OP	6E	ABS	3	6
ROR OP,X	7E	ABX	3	7

■ **RTI : return from interrupt**
nach Ausführen eines Interupt normales Programm weiter abarbeiten

Flags: N Z C I D V
wie vor Ausführung des Interrupts

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
RTI	40	–	1	6

■ **RTS : return from subroutine**
Rücksprung aus Unterprogramm

Flags: N Z C I D V
keine

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
RTS	60	–	1	6

■ **SBC : subtract with carry**
subtrahiere Speicherinhalt vom Akku unter Berücksichtigung des Vorzeichens

Flags:

N	Z	C	I	D	V
+	+	+			+

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
SBC #OP	E9	IM	2	2
SBC OP	E5	ZP	2	3
SBC OP,X	F5	ZPX	2	4
SBC OP	ED	ABS	3	4
SBC OP,X	FD	ABX	3	4
SBC OP,Y	F9	ABY	3	4
SBC (OP,X)	E1	(ZP,X)	2	6
SBC (OP),Y	F1	(ZP),Y	2	5

■ **SEC : set carry**
setze das Übertragsflag auf Eins

Flags:

N	Z	C	I	D	V
		+			

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
SEC	38	–	1	2

■ **SED : set decimal mode**
setze das Dezimal-Modus-Flag auf Eins

Flags:

N	Z	C	I	D	V
				+	

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
SED	F8	–	1	2

■ **SEI : set interrupt**
setze das Interruptflag auf Eins (es werden keine Interrupts mehr erlaubt)

Flags:

N	Z	C	I	D	V
			+		

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
SEI	78	–	1	2

■ **STA : store akku**
schreibe Akkuinhalt in Speicher

Flags: N Z C I D V
keine

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
STA OP	85	ZP	2	3
STA OP,X	95	ZPX	2	4
STA OP	8D	ABS	3	4
STA OP,X	9D	ABX	3	5
STA OP,Y	99	ABY	3	5
STA (OP,X)	81	(ZP,X)	2	6
STA (OP),Y	91	(ZP),Y	2	6

■ **STX : store X-register**
schreibe X-Registerinhalt in Speicher

Flags: N Z C I D V
keine

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
STX OP	86	ZP	2	3
STX OP,Y	96	ZPY	2	4
STX OP	8E	ABS	3	4

■ **STY : store Y-register**
schreibe Y-Registerinhalt in Speicher

Flags: N Z C I D V
keine

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
STY OP	84	ZP	2	3
STY OP,X	94	ZPX	2	4
STY OP	8C	ABS	3	4

Tabelle 2. Die Befehle des 6510-Prozessors (Fortsetzung)

■ TAX : transfer akku to X-register
schreibe Akkuinhalt ins X-Register

Flags: N Z C I D V
+ +

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
TAX	AA	–	1	2

■ TAY : transfer akku to Y-register
schreibe Akkuinhalt ins Y-Register

Flags: N Z C I D V
+ +

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
TAY	A8	–	1	2

■ TSX : transfer stackregister to X-register
schreibe Stackregisterinhalt ins X-Register

Flags: N Z C I D V
+ +

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
TSX	BA	–	1	2

■ TXA : transfer X-register to akku

Flags: N Z C I D V
+ +

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
TXA	8A	–	1	2

■ TXS : transfer X-register to stackregister
schreibe X-Registerinhalt ins Stackregister

Flags: N Z C I D V
keine

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
TXS	9A	–	1	2

■ TYA : transfer Y-register to akku
schreibe Y-Registerinhalt in Akku

Flags: N Z C I D V
+ +

Addressierungsarten:

Assembler:	Hex-Code:	Abkürzung:	Byte:	Takte:
TYA	98	–	1	2

Tabelle 2. Die Befehle des 6510-Prozessors (Schluß)

■ ABS : absolute (absolut)
Der Operand ist eine vierstellige, hexadezimale Adresse.

Beispiel:
LDA $C000
Der Inhalt der Adresse $C000 wird in den Akku geladen.

■ ABX : absolut X-indiziert
Der Operand ist eine vierstellige hexadezimale Zahl. Der Inhalt des X-Registers wird zum Operanden addiert und ergibt die Arbeits-Adresse.

Beispiel:
LDX #$10
LDA $C000,X
Der Inhalt der Speicherstelle $C010 ($C000 + $0010) wird in den Akku geladen.

■ ABY : absolut Y-indiziert
Der Operand ist eine vierstellige hexadezimale Zahl. Der Inhalt des Y-Registers wird zum Operanden addiert und ergibt die Arbeits-Adresse.

Beispiel:
LDY #$10
LDA $C000,Y
Der Inhalt der Speicherstelle $C010 ($C000 + $0010) wird in den Akku geladen.

>>Byte<< : In den Tabellen 2 und 3 gibt diese Spalte die jeweilige Länge des kompletten Befehls mit Operand an.

>>Flags<< : einzelne Bits des Statusregisters

N : negative flag. Zeigt an, daß bei einer Operation einer der beiden Operanden zwischen $80 (128) und $FF (255) liegt, also das letzte Bit gesetzt ist.

Z : zero flag. Zeigt an, daß das Ergebnis einer Operation im Akku gleich Null ist.

C : carry flag. Zeigt an, daß bei einer Operation ein Übertrag entstanden ist.

I : interrupt flag. Durch Setzen dieses Bit lassen sich Interrupts unterbinden.

D : decimal flag. Durch Setzen dieses Bit wir der Prozessor in den Dezimalmodus geschaltet. Das bedeutet, daß zum Beispiel das Ergebnis der Addition von $09 und $01 nicht $0A, sondern $10 ergibt.

V : overflow flag (Überlauf). Zeigt an, daß das Ergebnis einer Operation größer $FF (=255) war.

■ IM : immediate (unmittelbar)
Die Adressierungsart >>immediate<< bedeutet, daß der Operand unmittelbar als Wert weiterverarbeitet wird.

Beispiel:
LDA #$00
Die hexadezimale Zahl $00 wird direkt in den Akku geladen.

■ OP : Operand
Je nach Adressierungsart besteht der Operand eines Befehls aus einem (Adressierung >>immediate<< und >>zeropage<<) oder zwei Byte (>>absolute<<).

■ ZP : zeropage
Der Operand besteht aus einem Byte und gibt eine Adresse in der Zeropage (Speicherbereich $0000 bis $00FF) an.

Beispiel:
LDA $2B
Der Inhalt der Speicherstelle $002B wird in den Akku geladen.

■ ZPX : Zeropage X-indiziert
Der Inhalt des X-Registers wird zum zweistelligen, hexadezimalen Operanden addiert. Das Ergebnis ist eine Adresse in der Zeropage (Speicherbereich $0000 bis $00FF).

Beispiel:
LDX #$05
LDA $43,X
Der Inhalt der Adresse $0048 ($0043 + $0005) wird in den Akku geladen.

■ ZPY : Zeropage Y-indiziert
Der Inhalt des Y-Registers wird zum zweistelligen, hexadezimalen Operanden addiert. Das Ergebnis ist eine Adresse in der Zeropage (Speicherbereich $0000 bis $00FF).

Beispiel:
LDY #$05
LDA $43,Y
Der Inhalt der Adresse $0048 ($0043 + $0005) wird in den Akku geladen.

Tabelle 3. Diese Abkürzungen werden in den Tabellen 1 und 2 verwendet

■ (ZP,X) : indiziert indirekt
Der Inhalt des X-Registers wird zum zweistelligen, hexadezimalen Operanden addiert und ergibt eine Adresse in der Zeropage (Speicherbereich $0000 bis $00FF). Deren Inhalt und der Inhalt der darauffolgenden Adresse ergibt in der Form Lo-Byte/Hi-Byte die Arbeitsadresse.

Beispiel:

```
Adresse $20 hat den Inhalt $00
Adresse $21 hat den Inhalt $C0
LDX #$0E
LDA ($12,X)
```

Der Inhalt der Zeropage-Adressen $0020 ($000E + $0012) und $0021 ergibt die Arbeits-Adresse $C000. Deren Inhalt wird in den Akku geladen.

■ (ZP),Y : indirekt indiziert
Der zweistellige, hexdezimale Operand ergibt eine Adresse in der Zeropage (Speicherbereich $0000 bis $00FF). Deren Inhalt und der Inhalt der darauffolgenden Speicherstelle ergibt in der Form Lo-Byte/Hi-Byte eine Adresse, zu der der Inhalt des Y-Registers addiert wird. Das Ergebnis ist die Arbeitsadresse.

Beispiel:

```
Adresse $20 hat den Inhalt $00
Adresse $21 hat den Inhalt $C0
LDY #$10
LDA ($20),Y
```

Der Inhalt der Adresse $C010 ($C000 + $0010) wird in den Akku geladen.

Tabelle 3. Diese Abkürzungen werden in den Tabellen 1 und 2 verwendet (Schluß)

ROM-Routinen in eigenen Programmen

Das Rad ist schon erfunden! Ähnlich verhält es sich mit verschiedenen Routinen, die ein Assembler-Programmierer immer wieder benötigt. Aber warum soll man sich die Arbeit des Programmierens machen, wenn das Betriebssystem viele ständig benötigte Routinen schon enthält und man nur noch zu wissen braucht, ab welcher Adresse sie stehen?

Angenommen, Sie möchten in Assembler einige komplexe Dinge programmieren wie beispielsweise eine neue mathematische Funktion (wie wäre es mit dem Kotangens) und diese auf dem Bildschirm ausgeben. Das ist eine große Aufgabe, zu der zunächst einmal die Übernahme des Arguments in das Maschinenprogramm, dann einige Fließkomma-Rechenoperationen und schließlich die Ausgabe auf dem Bildschirm geschrieben werden müßten, wenn da nicht schon fast alles an verborgener Stelle als fertige Programm-Module im Computer vorhanden wäre!



Sowohl im unteren (von $A000 bis $BFFF) als auch im oberen ROM-Bereich (von $E000 bis $FFFF) liegt die Firmware fest verschachtelt vor. Der untere ROM-Abschnitt wird manchmal auch Basic-Interpreter, der obere ROM-Bereich Betriebssystem genannt, wobei diese Einteilung aber den Kern der Sache nicht genau trifft, denn Interpreter, Editor und Betriebssystem führen ein gemischtes Dasein quer durch alle genannten ROM-Bereiche hindurch.

Mindestens fünf Informationen braucht ein Assembler-Programmierer, wenn er das breite Programmangebot des ROMs nutzen möchte:

1. Einsprungadresse
2. Format der Eingabeparameter
3. Adressen der Eingabeparameter
4. Adressen der Ausgabeparameter
5. Format der Ausgabeparameter

Nicht alle Routinen, die man benutzen kann, erfordern alle fünf Informationen, manche weniger, einige auch mehr und schließlich gibt es noch Programmroutinen, die noch den Aufruf einer oder sogar mehrerer anderer Routinen nötig machen.

In der beigefügten Tabelle sind - nach Anwendungen sortiert - die wichtigsten Firmware-Möglichkeiten mit den erforderlichen Ein- und Ausgabeparametern aufgeführt. Das sind natürlich beileibe nicht alle. Die Auswahl erfolgte subjektiv! Es sind einfach diejenigen, die mir bislang am häufigsten untergekommen sind. Außerdem wurde auf die Kernel-Routinen verzichtet: Man findet diese sehr gut dokumentiert bereits in einer Reihe von Büchern und im Assembler-Kurs.

Die Tabelle nennt den Label-Namen, die Einsprungadresse und gibt eine Kurzbeschreibung der Funktion. Das Ein- und auch das Ausgabeformat ist ebenso angegeben wie auch die Adressen, an denen diese Parameter übergeben werden. Die verwendeten Bezeichnungen halten sich eng an die im Assembler-Kurs kennengelernten. Sie sind allgemein üblich:



FAC	Fließkomma-Akku 1
ARG	Fließkomma-Akku 2
A	Akkumulator
X,Y	X-, Y-Register
A/Y	2-Byte-Angabe im Format LSB/MSB im Akku/Y-Register
FLPT	Fließkommazahl im Normalformat
MFLPT	gepacktes Fließkommaformat

Damit das alles nicht so trocken abläuft, soll noch ein kleines Beispiel vorgestellt werden! Die oben schon erwähnte Kotangens-Funktion wird in einem Maschinenprogramm erzeugt, das durch USR anzuspringen ist. In Bild 1 finden Sie ein Flußdiagramm zu dem Programm, welches hier als Hypra-Ass-Listing abgebildet ist (Listing 1). Ein kurzes Testprogramm liefert Listing 2.

Der Einsprung mittels USR bietet den Vorteil, daß der Übergabewert gleich im FLPT-Format im FAC »landet«. Es ist aber sinnvoll, den Übergabeparameter mittels der MOVMF-Routine zu »retten«, weil durch die Kosinus-Funktion der FAC verändert wird. Wenn auch das Ergebnis der Kosinus-Funktion mittels MOVMF beiseite gelegt wurde, holen wir durch MOVFM den Anfangswert wieder in den FAC und bilden mittels SIN den Sinus davon. Schließlich teilen wir den im Speicher stehenden Kosinuswert durch den im FAC befindlichen Sinuswert (unter Verwendung von FDIV). Das Ergebnis ist der Kotangens:

COT X = (COS X/SIN X)

Dieser Wert befindet sich nun im FAC und wird mit dem RTS an das Basic-Programm zurückgeliefert. Im Testprogramm weisen wir ihm dann die Variable E zu.

Dieses kurze Beispiel soll Ihnen den Mund wässrig machen. Sehr viel detaillierter werden die ROM-Routinen im Kurs »Von Basic zu Assembler« im 64'er behandelt werden.

(Heino Ponnath/hm)

Literatur:
1. Kassera/Kassera, Programmieren in Maschinensprache, München 1985: Markt&Technik Verlag, MT 830
2. West, C 64 Computerhandbuch, München 1984, Te-wi
3. Babel/Krause/Dripke, Das Interface Age Systemhandbuch zum C 64, München 1983: Interface Age Verlag
4. Ponnath, C 64 Wunderland der Grafik, München 1985: Markt&Technik Verlag MT 756.

COT — Einsprung über USR-Vektor
Wert sichern durch MOVMF
COS
Ergebnis sichern durch MOVMF
Anfangswert zurück in FAC mittels MOVFM
SIN
Division des Cosinus durch den Sinus mit FDIV
RTS

Bild 1. Flußdiagramm einer Kotangens-Funktion

```
hypra-ass  assemblerlisting:

             10   -          .li 1,4,7
             20   -          .ba $6000
;
;einsprung mittels usr
;zuvor usr-vektor einstellen!
;
            160   -          .eq cos=$e264
            165   -          .eq movfm=$bba2
            170   -          .eq movmf=$bbd4
            180   -          .eq sin=$e26b
            190   -          .eq fdiv=$bb0f
            200   -          .eq wert=$7000
            205   -          .eq wert1=$7010
;
6000 a210   :212  -start     ldx #<(wert1)
6002 a070   :214  -          ldy #>(wert1)
6004 20d4bb :216  -          jsr movmf
6007 2064e2 :220  -          jsr cos
600a a200   :230  -          ldx #<(wert)
600c a070   :240  -          ldy #>(wert)
600e 20d4bb :250  -          jsr movmf
6011 a910   :252  -          lda #<(wert1)
6013 a070   :254  -          ldy #>(wert1)
6015 20a2bb :256  -          jsr movfm
6018 206be2 :260  -          jsr sin
601b a900   :270  -          lda #<(wert)
601d a070   :280  -          ldy #>(wert)
601f 200fbb :290  -          jsr fdiv
6022 60     :300  -          rts
;
            320   -          .sy 1,4,7

symbols in alphabetical order:

cos          = $e264
fdiv         = $bb0f
movfm        = $bba2
movmf        = $bbd4
sin          = $e26b
start        = $6000
wert         = $7000
wert1        = $7010

end of assembly  0:25.9
base = $6000  last byte at $6022
```

Listing 1. Hypra-Ass-Listing der Kotangens-Funktion

```
10 REM***TEST FUER COTANGENS***
20 POKE785,0:POKE786,96:REM USR-VEKTOR
30 INPUT"WINKEL";W:W=W*π/180:REM AUF BOGENMASS
40 E=USR(W):REM AUFRUF DES PROGRAMMES
50 PRINTW,E:REM ERGEBNIS IN E
60 END
READY.
```

Listing 2. Test der Kotangens-Funktion

1. Routinen, die die Kooperation von Basic und Assembler erleichtern:

Label	Adresse	Funktion	Eingabe Format	Eingabe Adresse	Ausgabe Format	Ausgabe Adresse
CHRGET	0073	Holt nächstes Byte	1 Byte	Basic-Text	1 Byte	A
CHRGOT	0073	Holt aktuelles Byte	1 Byte	Basic-Text	1 Byte	A
READY	A474	Erzeugt READY-Status	-	-	-	-
LINGET	A96B	Holt Integerwert (0-63999)	ASCII-Zahl	Basic-Text	2-Byte Integer	14/15
FRMNUM	AD8A	Holt beliebigen numerischen Ausdruck	Basic-Ausdruck	Basic-Text	FLPT	FAC
FRMEVL	AD9E	Holt beliebigen Ausdruck	Basic-Ausdruck	Basic-Text	a) bei Fließkomma: FLPT b) bei Integer: FLPT c) bei String: Zeiger auf Descriptor	FAC FAC FAC+3, FAC+4

Diese Routine setzt außerdem eine Reihe von Flaggen:
VALTYP($0D) 0=Zahl FF=String
INTFLAG ($0E) 0=Fließkomma 80=integer

		War Ausdruck einfache Variable, dann zeigt VARNAM ($45/6) das 1. Byte des Variablen-Namens				
CHKCLS	AEF7	Prüft auf ») «	ASCII	Basic-Text	-	-
CHKOPN	AEFA	Prüft auf » («	ASCII	Basic-Text	-	-
CHKCOM	AEFD	Prüft auf »,«	ASCII	Basic-Text	-	-
SYNCHR	AEFF	Prüft auf Zeichen im Akkumulator	ASCII	Basic-Text A	-	-
		Diese 4 Routinen überlesen das Zeichen, wenn vorhanden. Wenn nicht vorhanden, folgt SYNTAX ERROR				
ISVAR	AF28	Sucht Variablenwert	Name + Kennung	$45/46	a) Zahl: FLPT b) String: Descriptor-FAC+3	 FAC
ORDVAR	B0E7	Sucht Variablennamen	Name+ Kennung	$45/46	Adresse	$47/48
GTBYTC	B79B	Holt Zahl (0-255)	ASCII	Basic-Text	1 Byte	X
GETNUM	B7EB	Liest 2 Integerzahlen (Trennung durch Komma) 1. Zahl: 0 bis 65535 2. Zahl: 0 bis 255	ASCII	Basic-Text	2Byte-Int. 1Byte-Int.	$14/15 X
COMBYT	E200	Prüft auf »,« und holt folgende Zahl	ASCII	Basic-Text	1 Byte	X

2. Routinen, die Verschiebungen im Speicher durchführen:

BLTUC	A3BF	Verschiebt Blöcke	Adressen: Quelle Start Ende+1 Ziel Ende+1	 $5F/60 $5A/5B $58/59	-	-
PUTINT	A9C4	Schiebt FAC als Integer in Variable	FLPT Adresse	FAC $49/50	2Byte-Integer	angegebene Variable
PTFLPT	A9D6	Schiebt FAC in Variable	FLPT Adresse	FAC $49/50	MFLPT	angegebene Variable
GETSPT	AA2C	Schiebt String-descriptor in Variable	Zeiger Adresse	FAC+3 $49/50	Descriptor	angegebene Variable
STRVAL	B7B5	Zahlenstring in FAC einlesen	ASCII Länge	ab $22 A	FLPT	FAC
CONUPK	BA8C	Lädt ARG aus Speicher	MFLPT	A/Y	FLPT	ARG
MOVFM	BBA2	Lädt FAC aus Speicher	MFLPT	A/Y	FLPT	ARG
MOVMF	BBD4	Schiebt FAC in Speicher	FLPT Adresse	 FAC X/Y	MFLPT	angegebener Speicher
MOVFA	BBFC	ARG in FAC kopieren	FLPT	ARG	FLPT	FAC
MOVAF	BC0C	FAC in ARG kopieren	FLPT	FAC	FLPT	ARG
ACTOFC	BC3C	Akku in FAC schieben	1Byte	A	FLPT	FAC

3. Routinen zur Arithmetik:

ASCADD	AA27	Addiert ASCII-Ziffer zu FAC	ASCII Ziffer	A	FLPT	FAC
OROP	AFE6	FAC=(FAC)OR(ARG)	FLPT	FAC,ARG	FLPT	FAC
ANDOP	AFE9	FAC=(FAC)AND(ARG)	FLPT 0	FAC,ARG Y	FLPT	FAC
FACINX	B1AA	FAC wird als Integer in A/Y abgelegt	FLPT	FAC	2Byte-Integer	A/Y
UMULT	B357	16-Bit-Multiplikation	2-Byte-Integer Zahl1 Zahl2	 $28/29 $71/72	2Byte-Integer	X/Y
CIVAYF	B391	Integer (-32768 bis 32767) in FAC	2Byte-Integer	A/Y	FLPT	FAC
SGNFT	B3A2	Integer (0 bis 255) in FAC	1Byte	y	FLPT	FAC
GETADR	B7F7	Wandelt FAC zu Integer (0-65535)	FLPT	FAC	2Byte-Integer	Y/A +$14/15
FADDH	B849	FAC = FAC + 0,5	FLPT	FAC	FLPT	FAC
FSUB	B850	FAC=Speicherzahl -FAC	MFLPT FLPT	Zeiger A/Y FAC	FLPT	FAC
FSUBT	B853	FAC = ARG - FAC	FLPT	ARG,FAC	FLPT	FAC
FADD	B867	FAC=Speicherzahl +FAC	MFLPT MFLPT	Zeiger A/Y FAC	FLPT	FAC
FADDT	B86A	FAC = ARG + FAC	FLPT	ARG,FAC	FLPT	FAC
COMPLT	B947	Erzeugt Zweier-komplement von FAC	FLPT	FAC	FLPT	FAC
LOG	B9EA	FAC = ln(FAC)	FLPT	FAC	FLPT	FAC
FMULT	BA28	FAC=Speicherwert*FAC	MFLPT FLPT	Zeiger A/Y FAC	FLPT	FAC
FMULTT	BA30	FAC = ARG * FAC	FLPT	ARG,FAC	FLPT	FAC
MUL10	BAE2	FAC = 10 * FAC	FLPT	FAC	FLPT	FAC
DIV10	BAFE	FAC = FAC/10	FLPT	FAC	FLPT	FAC
FDIVF	BB07	FAC=ARG/Speicherzahl	MFLPT FLPT	Zeiger A/Y ARG	FLPT	FAC
FDIV	BB0F	FAC=Speicherzahl/FAC	MFLPT FLPT	Zeiger A/Y FAC	FLPT	FAC
FDIVT	BB14	FAC = ARG/FAC	FLPT	FAC,ARG	FLPT	FAC
SIGN	BC28	Ermittelt Vorzeichen von FAC	FLPT	FAC	1Byte 1 → + 0 → 0 FF → -	A
ABS	BC58	FAC = ABS(FAC)	FLPT	FAC	FLPT	FAC
FCOMP	BC5B	Vergleicht FAC mit Speicherzahl	MFLPT FLPT	Zeiger A/Y FAC	1Byte: 1: FAC > Speicher 0: FAC = Speicher FF: FAC < Speicher	A
INT	BCCC	FAC = INT(FAC)	FLPT	FAC	FLPT	FAC
AADD	BD7E	Addiert A zu FAC	FLPT 1Byte	FAC A	FLPT	FAC
SQR	BF71	FAC = SQR(FAC)	FLPT	FAC	FLPT	FAC
MPOT	BF78	FAC=Speicherwert ↑ FAC	FLPT MFLPT	FAC Zeiger A/Y	FLPT	FAC
FPWRT	BF7B	FAC = ARG ↑ FAC	FLPT	ARG,FAC	FLPT	FAC
NEGOP	BFB4	FAC = -FAC	FLPT	FAC	FLPT	FAC
EXP	BFED	FAC = e↑FAC	FLPT	FAC	FLPT	FAC
POLYX	E059	Polynomberechnung FAC=a0+a1x+a2x²+...	Adresse	Zeiger A/Y	FLPT	FAC
		Zeiger weist auf Start der Konstantentabelle. 1. Byte = Polynomgrad. Weitere Bytes sind die Koeffizienten des Polynoms in der Reihenfolge an,.....,a0 im MFLPT-Format.				
COS	E264	FAC = COS(FAC)	FLPT	FAC	FLPT	FAC
SIN	E26B	FAC = SIN(FAC)	FLPT	FAC	FLPT	FAC
TAN	E2B4	FAC = TAN(FAC)	FLPT	FAC	FLPT	FAC
ATN	E30E	FAC = ATN(FAC)	FLPT	FAC	FLPT	FAC

4. Auswahl von Ein-/Ausgabe-Routinen:

ERROR	A437	Fehlermeldung ausgeben und READY	Fehler-nummer	X	ASCII	Bildschirm
LIST	A69C	Listet Basic-Programm	-	-	-	-
NUMDON	AABC	Druckt FAC auf Bildschirm aus	FLPT	FAC	ASCII	Bildschirm
STROUT	AB1E	Gibt String auf Bildschirm aus. Ende=0	Adresse	Zeiger A/Y	ASCII	Bildschirm
SYNERR	AF08	Ausgabe SYNTAX ERROR	-	-	ASCII	Bildschirm
OVERR	B97E	Ausgabe OVERFLOW ERR.	-	-	ASCII	Bildschirm
LINPRT	BDCD	Druckt Integerzahl (0 bis 65535) aus.	2Byte-Integer	X/A	ASCII	Bildschirm
FACOUT	BDD7	Druckt FAC auf Bildschirm aus	FLPT	FAC	ASCII	Bildschirm
FOUT	BDDD	FAC wird zu ASCII-String (Ende=0). Kann direkt mit STROUT ausgegeben werden.	FLPT	FAC	ASCII	ab $100 (Ende=0) Startadr. A/Y
SAVET	E156	Save	Parameter aus Basic-Text			
VERFYT	E165	Verify	Parameter aus Basic-Text			
LOADT	E168	Load	Parameter aus Basic-Text			
SLPARA	E1D4	Holt Parameter für Save, Verify, Load aus dem Basic-Text				
PLOTK	E50A	Setzt Cursorposition	Zeile Spalte	X Y		
HOME	E566	Cursor in Home-Position				
PLOTR	E56C	Setzt Cursor-Position	Zeile Spalte	$D6 $D3		
GETKBC	E5B4	Holt Zeichen aus Tastaturpuffer	-	-	1Byte	A
PRT	E716	Gibt Zeichen in A auf Bildschirm aus	1Byte	A	ASCII	Bildschirm
CLRLN	E9FF	Löscht xte Bildschirmzeile	Zeilen-nummer	X	-	-

Tabelle der ROM-Routinen

Die Codes des C 64

Die kompletten ASCII-, CHR$- und Bildschirmcodes und dazu noch die vollständigen Tastencodes des C 64 finden Sie in der folgenden Tabelle anwenderfreundlich, also so, daß man sie auch benutzen kann, gegenübergestellt.

Programmieren in Maschinensprache beinhaltet oft Ein- und Ausgaben, zum Beispiel für eine Benutzerführung, oder um Eingaben vom Bildschirm einzulesen. Häufig steht man vor dem Problem, daß man nur einzelne Zeichen eines Textes in einem Maschinenprogramm verändern will. In allen Fällen benötigt man dann die CHR$- oder Bildschirmcodes, von denen hier die Rede ist. Das gleiche gilt natürlich auch für die Tastatur. Das Betriebssystem stellt uns die momentan gedrückte Taste in zwei Speicherstellen zur Verfügung.

Warum verschiedene Codes?

Vergleicht man in der Tabelle 1 die Spalten, so wird man feststellen, daß es einige nicht druckende Zeichen gibt, Codes, die auf dem Bildschirm nichts schreiben. Es sind dies die ASCII- beziehungsweise die CHR$-Codes von 0 bis 31 und von 128 bis 159. Auch findet man auf dem Bildschirm Zeichen, die in der ASCII-Tabelle nicht auftauchen – die reversen Zeichen mit den Codes von 128 bis 255. Um alle druckenden ASCII-Zeichen auch auf dem Bildschirm sichtbar zu machen, und dabei keine sinnlos leeren Felder mitzuführen, mußte man die Tabelle im Computer umstellen.

Ganz anders verhält es sich mit der Tastatur. Hier sieht die Codierung zunächst sehr zufällig aus. Natürlich ist dies mit der Anordnung der Tasten und der Methode, wie das Betriebssystem die Tasten abfragt, zu erklären. Uns soll es hier genügen, zu wissen, wie man von einem (Maschinen-) Programm aus einen Tastendruck registrieren kann. Man findet die zuletzt gedrückte Taste in den Speicherstellen 197 ($C5) und 203 ($CB), die mit PEEK beziehungsweise auf der Maschinensprache-Ebene mit LDA ausgelesen werden kann.

Die nachfolgende Tabelle 1 bietet auf einen Blick alle Codes, die im C 64 enthalten sind. Die Tabelle ist nach den CHR$-Zeichen sortiert, also eine aufsteigende Reihe von 0 bis 255. Um das Programmieren in Maschinensprache zu erleichtern, sind neben den dezimalen Zahlen auch die hexadezimalen Werte aufgeführt.

Die Spalten drei und vier enthalten die CHR$-Zeichen des C 64. Spalte drei zeigt dabei die Zeichen im Groß-/Grafikmodus, Spalte vier die entsprechenden Zeichen in Klein-/Großschriftmodus.

Die Zeichen von 0 bis 31 und von 128 bis 159 sind diejenigen CHR$-Codes, deren Aufruf auf dem Bildschirm kein sichtbares Zeichen hervorbringt, so zum Beispiel das Umstellen der Cursorfarbe, oder die Codes der Funktionstasten, die man nicht schreiben kann (?CHR$(133) ergibt nicht »F1«), aber für eine Abfrage (IF A$ = CHR$(133) THEN...) nutzen kann. Hier wird nicht zwischen den Modi unterschieden, da das Ergebnis in beiden Fällen das gleiche ist. Damit der Text in die Tabelle paßt, ließen wir ihn sogar bis in die sechste Spalte übergreifen. Über den Bildschirmcode kann man nicht über diese Funktionen verfügen!

Die Bildschirmcodes der fünften und sechsten Spalte sind die Werte, die man erhält, wenn man das CHR$-Zeichen mit PRINT ausgibt und die entsprechende Bildschirmspeicherstelle mit PEEK ausliest. Für Maschinensprache ist auch hier wieder der hexadezimale Wert angegeben. Die reversen Zeichen erhält man durch Addition von 128, in Maschinensprache durch den Befehl »ORA # $80«.

Obwohl der C 64 keinen echten ASCII-Zeichensatz hat, ist dieser in Spalte sieben aufgeführt. Man benötigt den ASCII-Code vor allem für die Datenfernübertragung (DFÜ) und zur Ansteuerung von nicht MPS-kompatiblen Druckern, die nicht mit einem »intelligenten« Interface betrieben werden. Die Erklärungen der verwendeten Kurzzeichen stehen in der Tabelle 2.

In den letzten beiden Spalten sind die Tastaturcodes enthalten, wie man sie beim Auslesen der Speicherstellen 197 und 203 erhält. Bitte beachten Sie, daß der Code 63 ($3F) der RUN/STOP-Taste entspricht, also dem CHR$-Code 3. Da aber PRINT CHR$(3) ein Programm nicht anhält und sich nicht abfragen läßt, ist es in der Tabelle nicht aufgeführt.

(og)

NUL	=	Null
SOH	=	Start of Heading
STX	=	Start of Text (Textbeginn)
ETX	=	End of Text (Textende)
EOT	=	End of Transmission (Übertragungsende)
ENQ	=	Enquiry (Testanfrage)
ACK	=	Acknowledge (Quittierung)
BEL	=	Bell (Klingelzeichen)
BS	=	Backspace (Zeichen zurück)
HT	=	Horizontal Tabulation
LF	=	Line Feed (Zeilenvorschub)
VT	=	Vertical Tabulation
FF	=	Form Feed (Formatanpassung)
CR	=	Carriage Return (Wagenrücklauf)
SO	=	Shift out (Shift aus)
SI	=	Shift in (Shift ein)
DLE	=	Data Link Escape (Datenverbindung abhalten)
DC	=	Device Control (Gerätesteuerung)
NAK	=	Negative Acknowledge (Gegenquittierung)
SYN	=	Synchronous Idle (synchronlos)
ETB	=	End of Transmission Block (Block Übertragungsende)
CAN	=	Cancel (Stornieren, zum Beispiel Puffer leeren)
EM	=	End of Medium (z. B. Papierende)
SUB	=	Substitute (Austausch)
ESC	=	Escape (übergehen, Druckercode)
FS	=	File Separator (Filetrennung)
GS	=	Group Separator (Trennung von Gruppen)
RS	=	Record Separator (Aufzeichnung trennen)
US	=	Unit Separator (Einheitentrenner)
SP	=	Space (Leerzeichen)
DEL	=	Delete (Löschen)

Tabelle 2. Erklärung der in Tabelle 1 abgekürzten ASCII-Zeichen

Wert dez	Wert hex	CHR$-Zeichen Groß/Grafik	CHR$-Zeichen Klein/Groß	Bildschirmcode dez	Bildschirmcode hex	ASCII-Code	Tastencode dez	Tastencode hex
0	00					NUL		
1	01					SOH		
2	02					STX		
3	03					ETX	63	3F
4	04					EOT		
5	05	weiß				ENQ		
6	06					ACK		
7	07					BEL		
8	08	blockiert SHIFT + C				BS		
9	09	entriegelt SHIFT + C				HT		
10	0A					LF		
11	0B					VT		
12	0C					FF		
13	0D	Carriage RETURN				CR	1	01
14	0E	Umschaltung auf Kleinschrift				SO		
15	0F					SI		
16	10					DLE		
17	11	Cursor down				DC1	7	07
18	12	Revers ein				DC2		
19	13	HOME (Cursor nach oben links)				DC3	51	33
20	14	DEL (ein Zeichen löschen)				DC4	0	00
21	15					NAK		
22	16					SYN		
23	17					ETB		
24	18					CAN		
25	19					EM		
26	1A					SUB		
27	1B					ESC		
28	1C	rot				FS		
29	1D	Cursor right				GS	2	02
30	1E	grün				RS		
31	1F	blau				US		
32	20			32	20	SP	60	3C
33	21	!	!	33	21	!		
34	22	"	"	34	22	"		
35	23	#	#	35	23	#		
36	24	$	$	36	24	$		
37	25	%	%	37	25	%		
38	26	&	&	38	26	&		
39	27	'	'	39	27	'		
40	28	(	(	40	28	(		
41	29	)	)	41	29	)		
42	2A	*	*	42	2A	*	49	31
43	2B	+	+	43	2B	+	40	28
44	2C	,	,	44	2C	,	47	2F
45	2D	–	–	45	2D	–	43	2B
46	2E	.	.	46	2E	.	44	2C
47	2F	/	/	47	2F	/	55	37
48	30	0	0	48	30	0	35	23
49	31	1	1	49	31	1	56	38
50	32	2	2	50	32	2	59	3B
51	33	3	3	51	33	3	8	08
52	34	4	4	52	34	4	11	0B
53	35	5	5	53	35	5	16	10
54	36	6	6	54	36	6	19	13
55	37	7	7	55	37	7	24	18
56	38	8	8	56	38	8	27	1B
57	39	9	9	57	39	9	32	20
58	3A	:	:	58	3A	:	45	2D
59	3B	;	;	59	3B	;	50	32
60	3C	<	<	60	3C	<		
61	3D	=	=	61	3D	=	53	35
62	3E	>	>	62	3E	>		
63	3F	?	?	63	3F	?		
64	40	@	@	0	00	@	46	2E
65	41	A	a	1	01	A	10	0A
66	42	B	b	2	02	B	28	1C
67	43	C	c	3	03	C	20	14
68	44	D	d	4	04	D	18	12
69	45	E	e	5	05	E	14	0E
70	46	F	f	6	06	F	21	15
71	47	G	g	7	07	G	26	1A
72	48	H	h	8	08	H	29	1D
73	49	I	i	9	09	I	33	21
74	4A	J	j	10	0A	J	34	22
75	4B	K	k	11	0B	K	37	25
76	4C	L	l	12	0C	L	42	2A
77	4D	M	m	13	0D	M	36	24
78	4E	N	n	14	0E	N	39	27
79	4F	O	o	15	0F	O	38	26
80	50	P	p	16	10	P	41	29
81	51	Q	q	17	11	Q	62	3E
82	52	R	r	18	12	R	17	11
83	53	S	s	19	13	S	13	0D
84	54	T	t	20	14	T	22	16
85	55	U	u	21	15	U	30	1E
86	56	V	v	22	16	V	31	1F
87	57	W	w	23	17	W	9	09
88	58	X	x	24	18	X	23	17
89	59	Y	y	25	19	Y	25	19
90	5A	Z	z	26	1A	Z	12	0C
91	5B	[	[	27	1B	[		
92	5C	£	£	28	1C	\	48	30
93	5D	]	]	29	1D	]		
94	5E	↑	↑	30	1E	↑	54	36
95	5F	←	←	31	1F	←	57	39
96	60	─	─	64	40	@		
97	61	♠	A	65	41	a		
98	62	│	B	66	42	b		
99	63	─	C	67	43	c		
100	64	─	D	68	44	d		
101	65	─	E	69	45	e		
102	66	─	F	70	46	f		
103	67	│	G	71	47	g		
104	68	│	H	72	48	h		
105	69	╮	I	73	49	i		
106	6A	╰	J	74	4A	j		
107	6B	╯	K	75	4B	k		
108	6C	└	L	76	4C	l		
109	6D	╲	M	77	4D	m		
110	6E	╱	N	78	4E	n		
111	6F	┌	O	79	4F	o		
112	70	┐	P	80	50	p		
113	71	●	Q	81	51	q		
114	72	─	R	82	52	r		
115	73	♥	S	83	53	s		
116	74	│	T	84	54	t		
117	75	╭	U	85	55	u		
118	76	╳	V	86	56	v		
119	77	○	W	87	57	w		
120	78	♣	X	88	58	x		
121	79	│	Y	89	59	y		
122	7A	♦	Z	90	5A	z		
123	7B	┼	┼	91	5B	[		
124	7C	▒	▒	92	5C	\		
125	7D	│	│	93	5D	]		
126	7E	π	▚	94	5E	↑		
127	7F	◥	▨	95	5F	DEL		
128	80							
129	81	orange						
130	82							
131	83							
132	84							
133	85	f1					4	04
134	86	f3					5	05
135	87	f5					6	06
136	88	f7					3	03
137	89	f2						
138	8A	f4						
139	8B	f6						
140	8C	f8						
141	8D	SHIFT RETURN						
142	8E	Umschaltung auf Groß/Grafik						
143	8F							
144	90	schwarz						
145	91	Cursor up						
146	92	Revers aus						
147	93	CLR (Bildschirm löschen)						
148	94	INSERT (Zeichen einfügen)						
149	95	braun						
150	96	hellrot						
151	97	grau 1						
152	98	grau 2						
153	99	hellgrün						
154	9A	hellblau						
155	9B	grau 3						
156	9C	lila						
157	9D	Cursor left						
158	9E	gelb						
159	9F	cyan						
160	A0			96	60			
161	A1	▌	▌	97	61			
162	A2	▄	▄	98	62			
163	A3	▔	▔	99	63			
164	A4	▁	▁	100	64			
165	A5	▏	▏	101	65			
166	A6	▒	▒	102	66			
167	A7	▕	▕	103	67			
168	A8	▒	▒	104	68			
169	A9	◤	▨	105	69			
170	AA	▕	▕	106	6A			
171	AB	├	├	107	6B			
172	AC	▗	▗	108	6C			
173	AD	└	└	109	6D			
174	AE	┐	┐	110	6E			
175	AF	▂	▂	111	6F			
176	B0	┌	┌	112	70			
177	B1	┴	┴	113	71			
178	B2	┬	┬	114	72			
179	B3	┤	┤	115	73			
180	B4	▏	▏	116	74			
181	B5	▎	▎	117	75			
182	B6	▐	▐	118	76			
183	B7	▔	▔	119	77			
184	B8	▀	▀	120	78			
185	B9	▃	▃	121	79			
186	BA	┘	✓	122	7A			
187	BB	▖	▖	123	7B			
188	BC	▝	▝	124	7C			
189	BD	┘	┘	125	7D			
190	BE	▘	▘	126	7E			
191	BF	▚	▚	127	7F			
192	C0	─	─	64	40			
193	C1	♠	A	65	41			
194	C2	│	B	66	42			
195	C3	─	C	67	43			
196	C4	─	D	68	44			

Tabelle 1. Die Codes des C 64



Wert		CHR$-Zeichen		Bildschirmcode		ASCII-Code	Tastencode	
dez	hex	Groß/ Grafik	Klein/ Groß	dez	hex		dez	hex
197	C5	─	E	69	45			
198	C6	─	F	70	46			
199	C7	│	G	71	47			
200	C8	│	H	72	48			
201	C9	╮	I	73	49			
202	CA	╰	J	74	4A			
203	CB	╯	K	75	4B			
204	CC	└	L	76	4C			
205	CD	╲	M	77	4D			
206	CE	╱	N	78	4E			
207	CF	┌	O	79	4F			
208	D0	┐	P	80	50			
209	D1	●	Q	81	51			
210	D2	─	R	82	52			
211	D3	♥	S	83	53			
212	D4	│	T	84	54			
213	D5	╭	U	85	55			
214	D6	╳	V	86	56			
215	D7	○	W	87	57			
216	D8	♣	X	88	58			
217	D9	│	Y	89	59			
218	DA	◆	Z	90	5A			
219	DB	┼	┼	91	5B			
220	DC	▒	▒	92	5C			
221	DD	│	│	93	5D			
222	DE	π	▒	94	5E			
223	DF	◥	▧	95	5F			
224	E0			96	60			
225	E1	▌	▌	97	61			
226	E2	▄	▄	98	62			
227	E3	▔	▔	99	63			
228	E4	▁	▁	100	64			
229	E5	▏	▏	101	65			
230	E6	▒	▒	102	66			
231	E7	▕	▕	103	67			
232	E8	▒	▒	104	68			
233	E9	◤	▧	105	69			
234	EA	▕	▕	106	6A			
235	EB	├	├	107	6B			
236	EC	▗	▗	108	6C			
237	ED	└	└	109	6D			
238	EE	┐	┐	110	6E			
239	EF	▂	▂	111	6F			
240	F0	┌	┌	112	70			
241	F1	┴	┴	113	71			
242	F2	┬	┬	114	72			
243	F3	┤	┤	115	73			
244	F4	▎	▎	116	74			
245	F5	▍	▍	117	75			
246	F6	▐	▐	118	76			
247	F7	▔	▔	119	77			
248	F8	─	─	120	78			
249	F9	▃	▃	121	79			
250	FA	┘	✓	122	7A			
251	FB	▖	▖	123	7B			
252	FC	▝	▝	124	7C			
253	FD	┘	┘	125	7D			
254	FE	▘	▘	126	7E			
255	FF	π	▒	94	5E			

Tabelle 1. Die Codes des C 64 (Schluß)

Befehlsübersicht – Zusammenfassung

Kennen Sie die Programme Hypra-Ass, Reassembler und SMON? Wenn ja, dann wird Ihnen die Befehlsübersicht bei der Arbeit mit diesen Programmen sehr hilfreich sein.

Selbst für den eingefleischten Profi ist es fast unmöglich alle Befehle von Hypra-Ass, Reassembler und SMON zu jeder Zeit parat zu haben. Um aber diese Programme optimal zu nutzen, ist die Kenntnis aller Befehle und deren Wirkungsweise sehr wichtig. Die nun folgende Befehlsübersicht soll Ihnen das ewige Nachschlagen ersparen. Natürlich kann es sich bei einer Übersicht nicht um eine ausführliche Anleitung handeln. Bei Verständnisschwierigkeiten finden Sie nähere Informationen an den entsprechenden Stellen in den Artikeln. (ak)

Quickreferenz Reassembler

P adresse	Einsprungspunkt durch Label markieren.
T adresse, adresse	Tabelle definieren.
E (byte)	Startet den Reassembler. Die einzelnen Bits des Bytes haben folgende Bedeutung:
Bit 0 gesetzt	Alle Zeropage-Adressen durch ein Label mit drei Buchstaben markieren.
Bit 1 gesetzt	Nach RTS, RTI, BRK, JMP Kommentarzeile einfügen.
Bit 2 gesetzt	Bei unmittelbarer Adressierung ASCII-Zeichen ausgeben.
Bit 3 gesetzt	Zwischen jede zweite Tabellenzeile Kommentarzeile einfügen.
Bit 4 gesetzt	Der ASCII-Ausdruck wird bei Tabellen unterdrückt.
Bit 5 gesetzt	Externe- und Tabellenlabel kennzeichnen.
Bit 6 gesetzt	Nach Tabellen suchen.
Bit 7 gesetzt	Speicherbereiche unter dem RAM reassemblieren.

Quickreferenz Hypra-Ass

Editorbefehle von Hypra-Ass

/A 100, 10	Automatische Zeilennumerierung. (Startzeile, Schrittweite)
/O	RENEW eines Quelltextes.
/D 100-200	Löschen von Zeilen und Zeilenbereichen.
/E 100-200	Listen von Zeilen und Zeilenbereichen.
TO,13;T1,24;T2,0;T3,10	Setzen von Tabulatoren. T0 = Tabulator für Assemblerbefehle. T1 = Tabulator für den Kommentar. T2 = Anzahl Blanks am Anfang einer Ausgabezeile. T3 = Tabulator für Symboltabelle.
/X	Verlassen des Assemblers.
/P1,100,200	Setzen eines Arbeitsbereichs (Page).
/ziffer (n)	Formatiertes Listen der Page.
/N1,100,10	Neu durchnumerieren einer Page mit Startnummer und Schrittweite.

Quickreferenz Hypra-Ass

Editorbefehle von Hypra-Ass

Befehl	Bedeutung
/F1, "string"	Suchen einer Zeichenkette in einer Page.
/R1, "string1", "string2"	String 2 wird innerhalb einer Page durch String 1 ersetzt.
/U 9000	Setzen des Quelltextstartes.
/B	Anzeige der aktuellen Speicherkonfiguration.
/S"name";/L"name";/V"name";/M"name"	Kurzform der Befehle SAVE, LOAD, VERIFY, MERGE.
/G9	Geräteadresse des Floppy-Laufwerks auf 9 umstellen.
/I	Lesen des Inhaltsverzeichnisses.
/K	Lesen des Fehlerkanals.
/@	Übermittlung von Diskettenbefehlen.
/CHO	Setzen der Hintergrundfarbe.
/CRO	Setzen der Rahmenfarbe.
/!	Ausgabe der Symboltabelle (unsortiert).
/!!	Ausgabe der Symboltabelle (sortiert).

Pseudo-Opcodes von Hypra-Ass

Befehl	Bedeutung
.BA adresse	Definiert Startadresse des Maschinenprogramms.
.EQ label = wert	Weist einem Label einen Wert zu.
.GL label = wert	Weist einem globalen Label einen Wert zu.
.BY 1,2,"a"	Einfügen von Byte-Werten in den Quelltext.
.WO 1234,label	Einfügen von Adressen in den Quelltext.
.TX "text"	Einfügen von Textblöcken in den Quelltext.
.AP "file"	Verketten von Quelltexten.
.OB "file,p,w"	Senden des Objektcodes zur Floppy.
.EN	Schließen des Objektfiles.
.ON ausdruck,sprung	Bedingter Sprung, wenn Ausdruck wahr.
.GO sprung	Unbedingter Sprung.
.IF ausdruck	Fortführung der Assemblierung bei .EL, falls Ausdruck falsch. Ansonsten hinter .IF bis zu .EL oder .EI.
.EL	ELSE Alternative zu den Zeilen, die hinter .IF stehen.
.EI	Ende der IF-Konstruktion.
.CO var1,var2	Übergabe von Labeln und Quelltext an nachgeladene Teile.
.MA makro (par1,par2)	Makrodefinitionszeile.
.RT	Ende der Makrodefinition.
...makro (par1,par2)	Makroaufruf.
.LI lfn,dn,ba	Senden von formatierten Listings (entspricht OPEN-Befehl).
.SY lfn,dn,ba	Senden der formatierten Symboltabelle.
.ST	Beendet die Assemblierung.
.DP t0,t1,t2,t3	Setzt die Tabulatoren aus dem Quelltext heraus.

Quickreferenz SMON. Die Klammern dürfen nicht mit eingegeben werden. Die Werte in den Klammern können, aber müssen nicht eingegeben werden.

Befehl	Bedeutung
A 4000	Zeilenassembler Startadresse = $4000.
B 4000 4200	Erzeugt Basic-DATA-Zeilen im Bereich $4000 bis $41FF
C 4010 4200 4013 4000 4200	Verschieben eines Programmes mit Adreßumrechnung. Entspricht W- und V-Befehl.
D 4000 (4100)	Disassembliert den Bereich von $4000 bis $4100
F	findet Zeichenketten (F), absolute Adressen (FA), relative Sprünge (FR), Tabellen (FT), Zeropage-Adressen (FZ) und Immediate-Befehle (FI).
GO 4000	Startet Maschinenprogramm (ab $4000)
IO 1	Ein-/Ausgabegerät auf Datasette umstellen
K A000 (A100)	Im angegebenen Bereich nach ASCII-Zeichen suchen.
L"name" (4000)	Laden eines Programmes an die richtige (oder angegebene) Adresse
M 4000 (4100)	Gibt den Inhalt des angegebenen Speicherbereichs als Hex-Byte und ASCII-Zeichen aus.
O 4000 4100 12	Füllt den angegebenen Bereich mit $12.
PO 5	Setzt Drucker-Geräteadresse auf 5
R	Registerinhalte anzeigen
S"name" 4000 4500	Speichert ein Programm von $4000 bis $4FFF.
TW (4000)	Einzelschrittmodus. Mit »J« können Unterprogramme in Echtzeit ausgeführt werden.
TB 4010 (05)	Breakpoint setzen (nach dem 5. Durchlauf)
TQ 4000	Schnellschrittmodus. Springt beim Erreichen eines Breakpoints in die Registeranzeige.
TS 4000 4020	Arbeitet ein Programm ab $4000 in Echtzeit ab und springt beim Erreichen von $4020 in die Registeranzeige.
V 6000 6200 4000 4100 4200	Ändert alle absoluten Adressen $4000 bis $41FF, die sich auf den Bereich $6000 bis $6200 beziehen, auf den neuen Bereich $4000.
W 4000 4300 5000	Verschiebt den Speicherinhalt von $4000 bis $42FF nach $5000.
X	Monitor verlassen.
# 49152	Dezimal umrechnen
$ 002B	Vierstellige Hex-Zahl umrechnen.
% 01100100	Achtstellige Binärzahl umrechnen.
? 0344+5234	Addition oder Subtraktion zweier vierstelliger Hex-Zahlen.
= 4000 5000	Vergleicht den Speicherinhalt von $4000 bis $5000.
Z	Ruft den Diskettenmonitor auf (falls implementiert). Dieser verfügt über folgende Befehle
R (12 01)	Liest Track $12 Sektor $01. Fehlt die Angabe hinter »R«, wird der logisch nächste Sektor gelesen.
W (12 01)	Schreibt Track $12 Sektor $01 auf Diskette. Fehlt die Angabe hinter »W«, werden die letzten Angaben von »R« benutzt.
M	Zeigt den Pufferinhalt als Hex-Dump.
X	Rücksprung zum Monitor.
F	Weitere Diskettenbefehle initialisieren (falls implementiert). Sind die Befehle initialisiert, stehen folgende Befehle zur Verfügung.
M (07)	Memory-Dump (Floppy-RAM/ROM) ausgeben.
V 6000 0400	Verschiebt einen 256-Byte-Block von $6000 ins Floppy-RAM nach $400.
@	Normale Diskettenbefehle senden
X	Zurück in normalen Diskettenmonitor.

Ist die Erweiterung »Neues vom SMON« implementiert, stehen folgende Befehle zur Verfügung:

Befehl	Bedeutung
Z 4000 (4100)	Gibt den Speicherinhalt von $4000 bis $40FF binär aus (ein Byte pro Zeile).
H 4000 (4100)	Gibt den Speicherbereich von $4000 bis $40FF binär aus (drei Byte pro Zeile).
N 4000 (4100)	Gibt den Speicherinhalt von $4000 bis $40FF im Bildschirmcode aus (32 Zeichen pro Zeile).
U 4000 (4100)	Wie »N« aber 40 Zeichen pro Zeile. Änderungen sind nicht möglich.
E 4000 (4100)	Füllt den Speicherbereich von $4000 bis $40FF mit $00.
Y 40	Verschiebt den SMON nach $4000.
Q 2000	Kopiert den Zeichensatz nach $2000.
J	Bringt letzten Ausgabebefehl zurück.

Impressum

Herausgeber: Carl-Franz von Quadt, Otmar Weber

Chefredakteur: Michael Scharfenberger

Leitender Redakteur: Albert Absmeier

Redaktion: Volker Everts, Achim Hübner, Georg Klinge (Koordination), Harald Meyer, Markus Ohnesorg, Thomas Röder, Boris Schneider, Arnd Wängler

Fremdautoren:
Ponnath, Müller, Lonczewski, Weineck, Möllmann, Mann, Zwartscholten, Meyer, Heister, Pasch, Sajonz, Mootz, Krätzig, Bauschke, Reiser, Bauer, Reetz, Rosen, Schramm, Sauer, Richters

Layout: Leo Eder (Ltg.)

Herstellung: Klaus Buck

Auslandsrepräsentation:

Schweiz: Markt & Technik Vertriebs AG, Kollerstr. 3, CH-6300 Zug, Tel. 042-41 56 56, Telex: 862 329

USA: M&T Publishing Inc.; 2464 Embarcadero Way, Palo Alto, CA 94 303

Manuskripteinsendungen: Manuskripte und Programmlistings werden gerne von der Redaktion angenommen. Sie müssen frei sein von Rechten Dritter. Sollten sie auch an anderer Stelle zur Veröffentlichung oder gewerblichen Nutzung angeboten werden, so muß dies angegeben werden. Mit der Einsendung von Manuskripten und Listings gibt der Verfasser die Zustimmung zum Abdruck in von der Markt & Technik Verlag AG herausgegebenen Publikationen und zur Vervielfältigung der Programmlistings auf Datenträger. Mit der Einsendung von Bauanleitungen gibt der Einsender die Zustimmung zum Abdruck in von Markt & Technik Verlag AG verlegten Publikationen und dazu, daß Markt & Technik Verlag AG Geräte und Bauteile nach der Bauanleitung herstellen läßt und vertreibt oder durch Dritte vertreiben läßt. Honorare nach Vereinbarung. Für unverlangt eingesandte Manuskripte und Listings wird keine Haftung übernommen.

Leitung Marketing Vertrieb: Hans Hörl (114)

Vertriebsleitung: Helmut Grünfeldt (189)

Anzeigenverwaltung und Disposition: Michaela Hörl

Verlagsleiter M&T-Buchverlag: Günther Frank

Druck: St. Otto Verlag GmbH, Laubanger 23, 8600 Bamberg

Preis: Das Einzelheft kostet DM 14,—

Vertrieb Handelsauflage: Inland (Groß-, Einzel- und Bahnhofsbuchhandel) sowie Österreich und Schweiz: Pegasus Buch- und Zeitschriften-Vertriebs GmbH, Hauptstätter Straße 96, 7000 Stuttgart 1, Telefon (07 11) 6 48 30

Verantwortlich:
Für redaktionellen Teil: Michael Scharfenberger
Für Anzeigen: Brigitte Fiebig

Redaktions-Direktor: Michael M. Pauly

Vorstand: Carl-Franz von Quadt, Otmar Weber

Anschrift für Verlag, Redaktion, Vertrieb, Anzeigenverwaltung und alle Verantwortlichen:
Markt & Technik Verlag Aktiengesellschaft,
Hans-Pinsel-Straße 2, 8013 Haar bei München,
Telefon (0 89) 46 13-0, Telex 5-22 052





